

Using Microsoft Power BI with AllegroGraph

There are multiple methods to integrate AllegroGraph SPARQL results into Microsoft Power BI. In this document we describe two best practices to automate queries and refresh results if you have a production AllegroGraph database with new streaming data:

The first method uses Python scripts to feed Power BI. The second method issues SPARQL queries directly from Power BI using POST requests.

Method 1: Python Script:

Assuming you know Python and have it installed locally, this is definitely the easiest way to incorporate SPARQL results into Power BI. The basic idea of the method is as follows: First, the Python script enables a connection to your desired AllegroGraph repository. Then we utilize AllegroGraph's Python API within our script to run a SPARQL query and return it as a Pandas dataframe. When running this script within Power BI Desktop, the Python scripting service recognizes all unique dataframes created, and allows you to import the dataframe into Power BI as a table, which can then be used to create visualizations.

Requirements:

1. You must have the AllegroGraph Python API installed. If you do not, installation instructions are here: <https://franz.com/agraph/support/documentation/current/python/install.html>
2. Python scripting must be enabled in Power BI Desktop. Instructions to do so are here: <https://docs.microsoft.com/en-us/power-bi/connect-data/desktop-python-scripts>

a) As mentioned in the article, pandas and matplotlib must be installed. This can be done with 'pip install pandas' and 'pip install matplotlib' in your terminal.

The Process:

Once these requirements have been met, create a Python file with whatever script editor you usually use. The following code will create a connection to your desired repository. For this example, we will be using the Kennedy dataset that is available with the AllegroGraph distribution (See the 'Tutorial' directory). Load the Kennedy.ntriples file into your running AllegroGraph. (Replace the '****' in the code with your corresponding username and password.)

#the necessary imports

```
import os

from franz.openrdf.connect import ag_connect

from franz.openrdf.query.query import QueryLanguage

import pandas as pd
```

#connect to your agraph repository

```
def setup_env_var(var_name, value, description):

    os.environ[var_name] = value

    print("{}: {}".format(description, value))

setup_env_var('AGRAPH_HOST', 'localhost', 'Hostname')

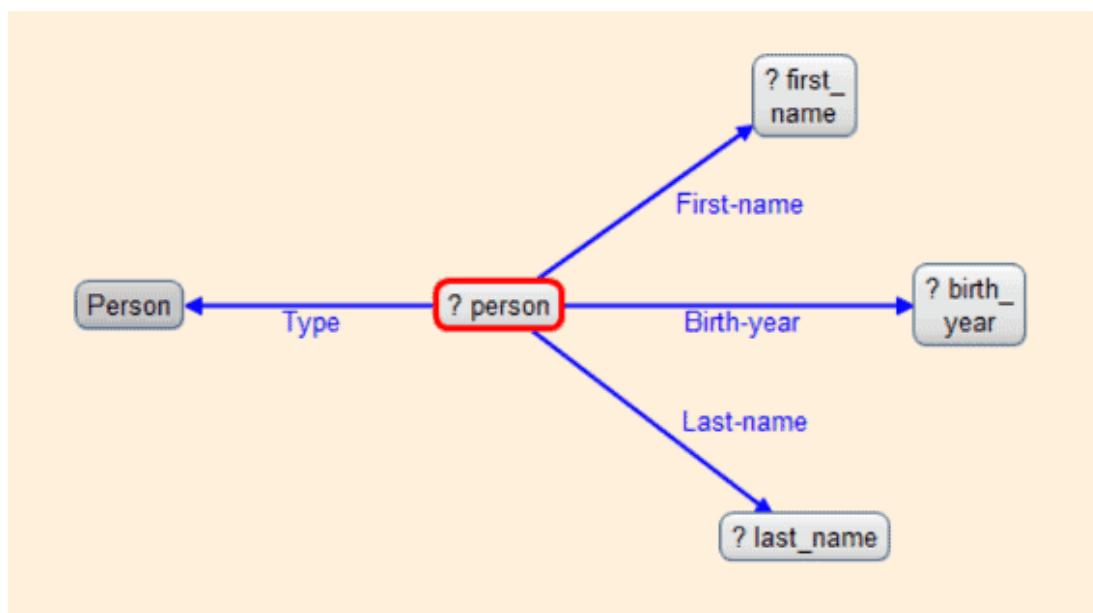
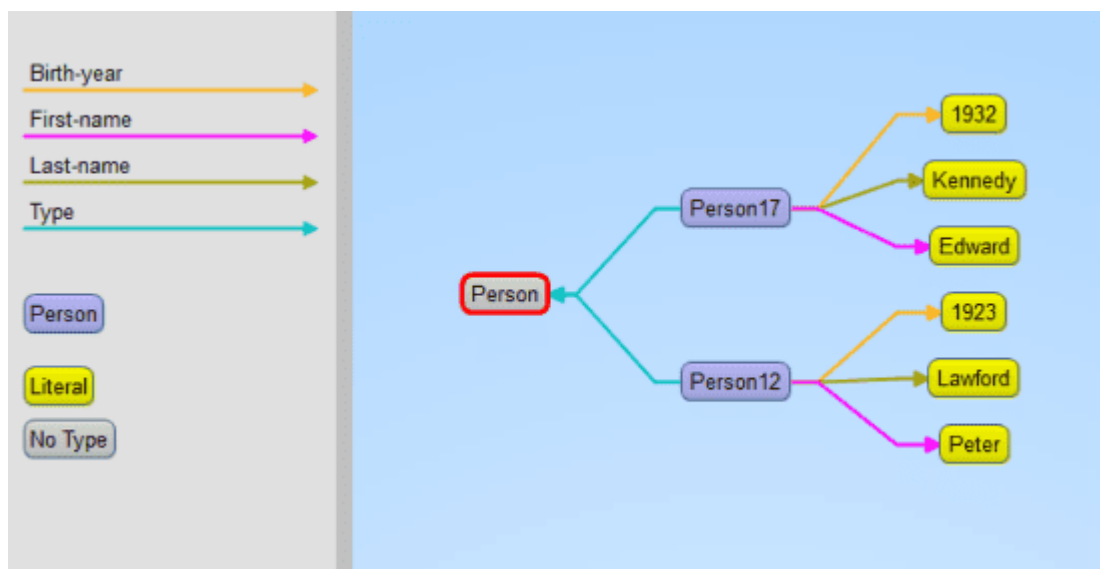
setup_env_var('AGRAPH_PORT', '10035', 'Port')

setup_env_var('AGRAPH_USER', '****', 'Username')
```

```
setup_env_var('AGRAPH_PASSWORD', '****', 'Password')
```

```
conn = ag_connect('kennedy', create=False, clear=False)
```

2. We then want to create a query. For this example, we will first show what our data looks like, what the visual query of the information is, and what the written query looks like. With the following query we want every person's first and last names, as well as their birth years. Here is a small portion of the data visualized in Gruff, and then the visualization of the query:

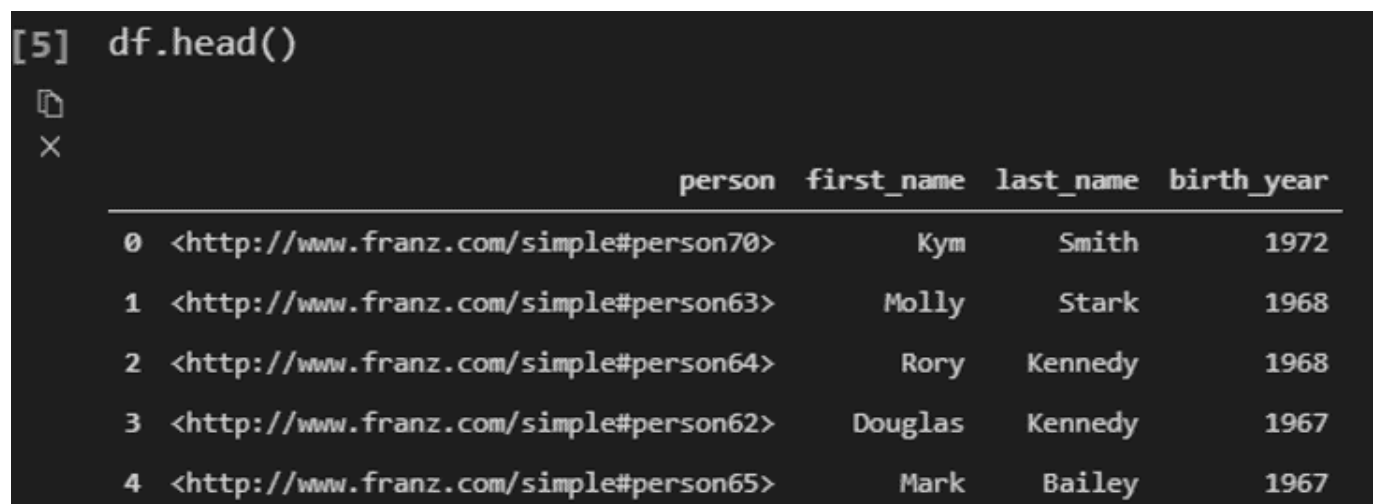


3. Then add the written query to the python script as a variable string (we added an additional line to the query to sort on birth year). Next use the API functionality to simply execute the query and turn the results into a pandas dataframe.

```
query = """select ?person ?first_name ?last_name ?birth_year
where
{ ?person <http://www.franz.com/simple#first-name> ?first_name
;
      <http://www.franz.com/simple#birth-year> ?birth_year
;
      rdf:type <http://www.franz.com/simple#person> ;
      <http://www.franz.com/simple#last-name> ?last_name .
}
order by desc(?birth_year)"""

with conn.executeTupleQuery(query) as result:
    df = result.toPandas()
```

When looking at the result, we see that we have a DataFrame!

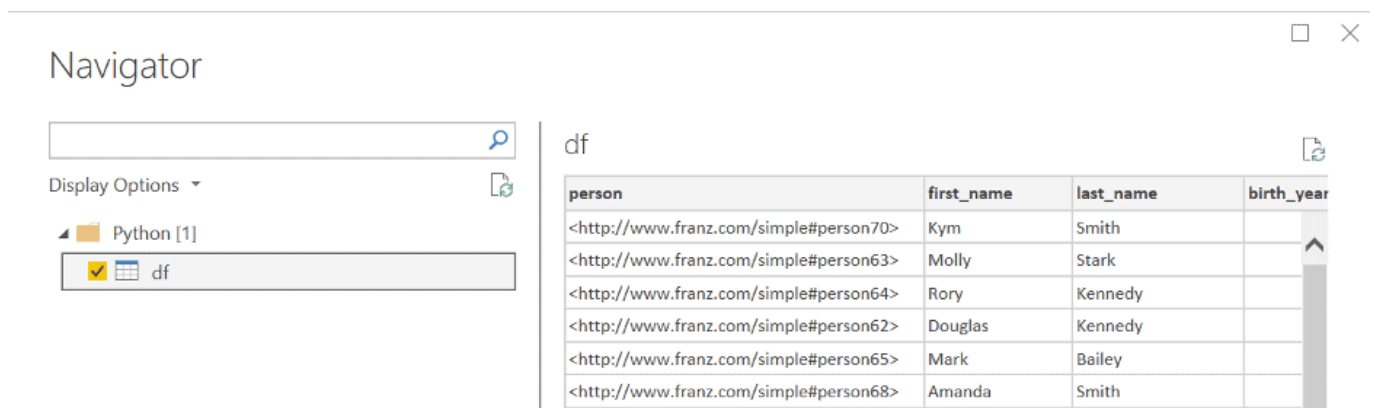


```
[5] df.head()
```

	person	first_name	last_name	birth_year
0	<http://www.franz.com/simple#person70>	Kym	Smith	1972
1	<http://www.franz.com/simple#person63>	Molly	Stark	1968
2	<http://www.franz.com/simple#person64>	Rory	Kennedy	1968
3	<http://www.franz.com/simple#person62>	Douglas	Kennedy	1967
4	<http://www.franz.com/simple#person65>	Mark	Bailey	1967

4. Now we will use this script in Power BI. When in Power BI Desktop, go to 'Get Data' and look for the python script option. Then simply copy and paste your entire script into the

text box, and run the script. In this case, our output looks like this:



5. Next simply 'Load' the data, and then you can use the Power BI Desktop interface to create whatever visualizations you want! If you do have a lot of additional operations to perform on your dataframe, we recommend doing these in your python script.

Method 2: POST Request:

For the SPARQL query via POST requests to work you need to url-encode the query. Every modern programming language will support that, but in our example we will be using Python again. This method is better for when you do not have python locally installed or prefer a different programming language.

It is possible to send a GET request from Power BI, but once the results from the query reach a certain size, a POST request is required, which is confusing to do within the Power BI Desktop interface. The following steps will show you how to do SPARQL Queries using POST requests. It looks a bit odd but it works well.

The Process:

1. In your AG WebView create an 'anonymous' user. (Go to admin -> Users -> [add a user] -> and add 'anonymous' as

username without adding a password). You can use these settings:

Users

anonymous [\[remove\]](#)

Roles: None

[\[suspend\]](#) [\[disable\]](#) [\[expire password\]](#)

☐ Superuser ☐ Start sessions ☐ Evaluate arbitrary code ☐ Control replication ☐ Two-phase commit

☒ Allow user attributes via HTTP header `x-user-attributes`

☐ Allow user attributes via SPARQL PREFIX `franzOption_userAttributes`

◦ `read/write` on all [\[remove\]](#)

Grant on catalog repository [\[ok\]](#)

Security Filters: `None` [\[add\]](#)

2. Go to your desired repository in WebView and Click on 'Queries' -> 'New'

3. Write a simple SPARQL query, and run it to make sure you get the correct response back.

4. In python create the following script: (Assuming your AllegroGraph is on your localhost port 10035 and your repo is called 'kennedy')

```
import urllib
```

```
def CreatePOSTquery(query):
```

```
start =
```

```
"http://anonymous:@localhost:10035/repositories/kennedy?queryLimit=SPARQL&limit=1000&infer=false&returnQueryMetadata=false&checkVariables=false&query="
```

```
    response = start + urllib.parse.quote(query)
```

```
    return response
```

This function url-encodes the query and attaches it to the POST request. Replace the 'localhost:10035' and 'kennedy' strings in the start variable with your corresponding data. Then, using the same query as our previous example, we create

our url-encoded POST query:

```
query = """select ?person ?first_name ?last_name ?birth_year
where
{ ?person <http://www.franz.com/simple#first-name> ?first_name
;
    <http://www.franz.com/simple#birth-year> ?birth_year
;
    rdf:type <http://www.franz.com/simple#person> ;
    <http://www.franz.com/simple#last-name> ?last_name .
}
order by desc(?birth_year)"""
```

```
result = CreatePOSTquery(query)
print(result)
```

This gives us the following result:

```
[16] result
```

```
'http://anonymous:@localhost:10035/repositories/kennedy?queryLn=SPARQL&limit=1000&infer=false&returnQueryMetadata=false&checkVariables=false&query=select%20%3Fperson%20%3Ffirst_name%20%3Flast_name%20%3Fbirth_year%20where%0A%7B%20%3Fperson%20%3Chttp%3A/www.franz.com/simple%23first-name%3E%20%3Ffirst_name%20%38%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%3Chttp%3A/www.franz.com/simple%23birth-year%3E%20%3Fbirth_year%20%38%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20rdx%3Atype%20%3Chttp%3A/www.franz.com/simple%23person%3E%20%38%0A%20%20%20%20%20%20%20%20%20%3Chttp%3A/www.franz.com/simple%23last-name%3E%20%3Flast_name%20.%20%7D%0Aorder%20by%20desc%28%3Fbirth_year%29'
```

5. Within Power BI Desktop we go to 'Get data' and create a 'Blank query' and go into the 'Advanced Editor' window. Using the following format we will get our desired results (please note that due to the length of the url-encoded request, it did not all fit in the image. Copy and pasting into the url field works fine. The 'url' variable needs to be in quotes and have a comma at the end):

Query1

```

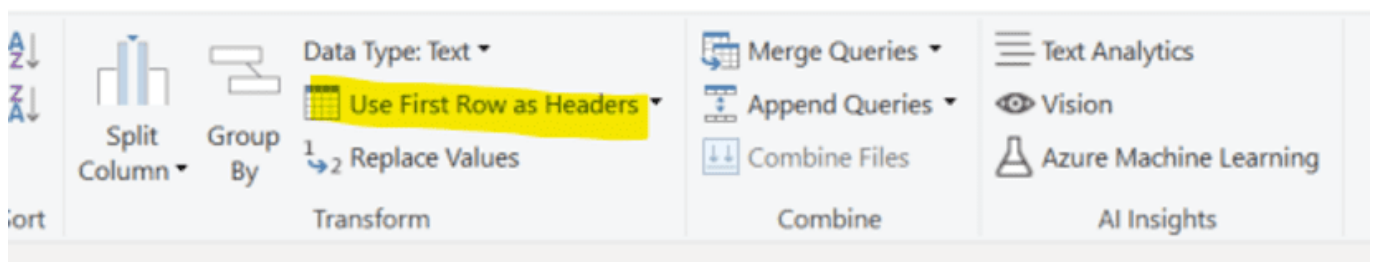
let
    url = "http://anonymous:@localhost:10035/repositories/kennedy?queryLn=SPARQL&limit=1000&infer=false&returnQuery",
    body = "",
    Source = Csv.Document(Web.Contents(url), [Headers = [Accept="text/csv"], Content=Text.ToBinary(body)]])
in
    Source

```

We see the following results:

Column1	Column2	Column3	Column4
person	first_name	last_name	birth_year
http://www.franz.com/simple#person70	Kym	Smith	1972
http://www.franz.com/simple#person63	Molly	Stark	1968
http://www.franz.com/simple#person64	Rory	Kennedy	1968
http://www.franz.com/simple#person62	Douglas	Kennedy	1967
http://www.franz.com/simple#person65	Mark	Bailey	1967
http://www.franz.com/simple#person68	Amanda	Smith	1967
http://www.franz.com/simple#person71	Alfred	Tucker	1967
http://www.franz.com/simple#person76	Patrick	Kennedy	1967
http://www.franz.com/simple#person23	Carolyn	Bessette	1966
http://www.franz.com/simple#person69	Cart	Hood	1966
http://www.franz.com/simple#person32	Jeannie	Ripp	1965
http://www.franz.com/simple#person33	Anthony	Shriver	1965
http://www.franz.com/simple#person34	Alina	Mojica	1965
http://www.franz.com/simple#person60	Matthew	Kennedy	1965
http://www.franz.com/simple#person31	Mark	Shriver	1964
http://www.franz.com/simple#person61	Victoria	Stauss	1964
http://www.franz.com/simple#person24	Patrick	Kennedy	1963
http://www.franz.com/simple#person58	Christopher	Kennedy	1963

6. One last step is to turn the top row into the column names, which can be achieved by pressing the 'Use first row as headers':



The best part about both of these methods is that once the query has been created, Power BI can refresh the visuals using the same queries if your data changed. This can be achieved by

scheduling refreshes within the Power BI Desktop interface (<https://docs.microsoft.com/en-us/power-bi/connect-data/refresh-data#configure-scheduled-refresh>)

Please send any questions or issues to: support@franz.com

Using JSON-LD in AllegroGraph – Python Example



The following is example #19 from our AllegroGraph Python Tutorial.

JSON-LD is described pretty well at <https://json-ld.org/> and the specification can be found at <https://json-ld.org/latest/json-ld/>.

The website <https://json-ld.org/playground/> is also useful.

There are many reasons for working with JSON-LD. The major search engines such as Google require ecommerce companies to mark up their websites with a systematic description of their products and more and more companies use it as an easy serialization format to share data.

The benefit for your organization is that you can now combine your documents with graphs, graph search and graph algorithms. Normally when you store documents in a document store you set up your documents in such a way that it is optimized for direct retrieval queries. Doing complex joins for multiple

types of documents or even doing a shortest path through a mass of object (types) is however very complicated. Storing JSON-LD objects in AllegroGraph gives you all the benefits of a document store *and* you can semantically link objects together, do complex joins and even graph search.

A second benefit is that, as an application developer, you do not have to learn the entire semantic technology stack, especially the part where developers have to create individual triples or edges. You can work with the JSON data serialization format that application developers usually prefer.

In the following you will first learn about JSON-LD as a syntax for semantic graphs. After that we will talk more about using JSON-LD with AllegroGraph as a document-graph-store.

Setup

You can use Python 2.6+ or Python 3.3+. There are small setup differences which are noted. You do need *agraph-python-101.0.1* or later.

Mimicking instructions in the Installation document, you should set up the virtualenv environment.

1. Create an environment named *jsonld*:

```
python3 -m venv jsonld
```

or

```
python2 -m virtualenv jsonld
```

2. Activate it:

Using the Bash shell:

```
source jsonld/bin/activate
```

Using the C shell:

```
source jsonld/bin/activate.csh
```

3. Install *agraph-python*:

```
pip install agraph-python
```

And start **python**:

```
python  
[various startup and copyright messages]  
>>>
```

We assume you have an AllegroGraph 6.5.0 server running. We call **ag_connect**. Modify the *host*, *port*, *user*, and *password* in your call to their correct values:

```
from franz.openrdf.connect import ag_connect  
with ag_connect('repo', host='localhost', port='10035',  
                user='test', password='xyzzzy') as conn:  
    print (conn.size())
```

If the script runs successfully a new repository named *repo* will be created.

JSON-LD setup

We next define some utility functions which are somewhat different from what we have used before in order to work better with JSON-LD. **createdb()** creates and opens a new repository and **opendb()** opens an existing repo (modify the values of *host*, *port*, *user*, and *password* arguments in the definitions if necessary). Both return repository connections which can be used to perform repository operations. **showtriples()** displays triples in a repository.

```

import os
import json, requests, copy

from franz.openrdf.sail.allegrographserver import
AllegroGraphServer
from franz.openrdf.connect import ag_connect
from franz.openrdf.vocabulary.xmlschema import XMLSchema
from franz.openrdf.rio.rdfformat import RDFFormat

# Functions to create/open a repo and return a
RepositoryConnection
# Modify the values of HOST, PORT, USER, and PASSWORD if
necessary

def createdb(name):
    return
    ag_connect(name,host="localhost",port=10035,user="test",passwo
rd="xyzy",create=True,clear=True)

def opendb(name):
    return
    ag_connect(name,host="localhost",port=10035,user="test",passwo
rd="xyzy",create=False)

def showtriples(limit=100):
    statements = conn.getStatements(limit=limit)
    with statements:
        for statement in statements:
            print(statement)

```

Finally we call our **createdb** function to create a repository and return a *RepositoryConnection* to it:

```
conn=createdb('jsonplay')
```

Some Examples of Using JSON-LD

In the following we try things out with some JSON-LD objects that are defined in json-ld playground: jsonld

The first object we will create is an *event dict*. Although it is a Python dict, it is also valid JSON notation. (But note that not all Python dictionaries are valid JSON. For example, JSON uses null where Python would use None and there is no magic to automatically handle that.) This object has one key called @context which specifies how to translate keys and values into predicates and objects. The following @context says that every time you see ical: it should be replaced by <http://www.w3.org/2002/12/cal/ical#>, xsd: by <http://www.w3.org/2001/XMLSchema#>, and that if you see ical:dtstart as a key then the value should be treated as an xsd:dateTime.

```
event = {
    "@context": {
        "ical": "http://www.w3.org/2002/12/cal/ical#",
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "ical:dtstart": { "@type": "xsd:dateTime" }
    },
    "ical:summary": "Lady Gaga Concert",
    "ical:location": "New Orleans Arena, New Orleans,
Louisiana, USA",
    "ical:dtstart": "2011-04-09T20:00:00Z"
}
```

Let us try it out (the subjects are blank nodes so you will see different values):

```
>>> conn.addData(event)
>>> showtriples()
(_:b197D2E01x1, <http://www.w3.org/2002/12/cal/ical#summary>,
"Lady Gaga Concert")
(_:b197D2E01x1, <http://www.w3.org/2002/12/cal/ical#location>,
"New Orleans Arena, New Orleans, Louisiana, USA")
(_:b197D2E01x1, <http://www.w3.org/2002/12/cal/ical#dtstart>,
"2011-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
```

Adding an @id and @type to Objects

In the above we see that the JSON-LD was correctly translated into triples but there are two immediate problems: first each subject is a blank node, the use of which is problematic when linking across repositories; and second, the object does not have an RDF type. We solve these problems by adding an @id to provide an IRI as the subject and adding a @type for the object (those are at the lines just after the @context definition):

```
>>> event = {
    "@context": {
        "ical": "http://www.w3.org/2002/12/cal/ical#",
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "ical:dtstart": { "@type": "xsd:dateTime" }
    },
    "@id": "ical:event-1",
    "@type": "ical:Event",
    "ical:summary": "Lady Gaga Concert",
    "ical:location": "New Orleans Arena, New Orleans,
Louisiana, USA",
    "ical:dtstart": "2011-04-09T20:00:00Z"
}
```

We also create a test function to test our JSON-LD objects. It is more powerful than needed right now (here we just need *conn, addData(event)* and *showTriples()* but **test** will be useful in most later examples. Note the *allow_external_references=True* argument to *addData()*. Again, not needed in this example but later examples use external contexts and so this argument is required for those.

```
def
test(object, json_ld_context=None, rdf_context=None, maxPrint=100
, conn=conn):
    conn.clear()
    conn.addData(object, allow_external_references=True)
    showtriples(limit=maxPrint)
```

```
>>> test(event)
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/2002/12/cal/ical#summary>, "Lady Gaga
Concert")
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/2002/12/cal/ical#location>, "New Orleans
Arena, New Orleans, Louisiana, USA")
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/2002/12/cal/ical#dtstart>,
"2011-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://www.w3.org/2002/12/cal/ical#event-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://www.w3.org/2002/12/cal/ical#Event>)
```

Note in the above that we now have a proper subject and a type.

Referencing a External Context Via a URL

The next object we add to AllegroGraph is a person object. This time the @context is not specified as a JSON object but as a link to a context that is stored at <http://schema.org/>. Also in the definition of the function test above we had this parameter in `addData:allow_external_references=True`. Requiring that argument explicitly is a security feature. One should use external references only that context at that URL is trusted (as it is in this case).

```
person = {
    "@context": "http://schema.org/",
    "@type": "Person",
    "@id": "foaf:person-1",
    "name": "Jane Doe",
    "jobTitle": "Professor",
    "telephone": "(425) 123-4567",
    "url": "http://www.janedoe.com"
}
```

```
>>> test(person)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/name>, "Jane Doe")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/jobTitle>, "Professor")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/telephone>, "(425) 123-4567")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/url>, <http://www.janedoe.com>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://schema.org/Person>)
```

Improving Performance by Adding Lists

Adding one person at a time requires doing an interaction with the server for each person. It is much more efficient to add lists of objects all at once rather than one at a time. Note that `addData` will take a list of dicts and still do the right thing. So let us add a 1000 persons at the same time, each person being a copy of the above person but with a different `@id`. (The example code is repeated below for ease of copying.)

```
>>> x = [copy.deepcopy(person) for i in range(1000)]
>>> len(x)
1000
>>> c = 0
>>> for el in x:
    el['@id'] = "http://franz.com/person-" + str(c)
    c = c + 1
>>> test(x,maxPrint=10)
(<http://franz.com/person-0>, <http://schema.org/name>, "Jane
Doe")
(<http://franz.com/person-0>, <http://schema.org/jobTitle>,
"Professor")
(<http://franz.com/person-0>, <http://schema.org/telephone>,
"(425) 123-4567")
(<http://franz.com/person-0>, <http://schema.org/url>,
```



```

<http://www.janedoe.com>)
(<http://franz.com/person-0>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://schema.org/Person>)
(<http://franz.com/person-1>, <http://schema.org/name>, "Jane
Doe")
(<http://franz.com/person-1>, <http://schema.org/jobTitle>,
"Professor")
(<http://franz.com/person-1>, <http://schema.org/telephone>,
"(425) 123-4567")
(<http://franz.com/person-1>, <http://schema.org/url>,
<http://www.janedoe.com>)
(<http://franz.com/person-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://schema.org/Person>)
>>> conn.size()
5000
>>>

```

```

x = [copy.deepcopy(person) for i in range(1000)]
len(x)

c = 0
for el in x:
    el['@id'] = "http://franz.com/person-" + str(c)
    c = c + 1

test(x,maxPrint=10)

conn.size()

```

Adding a Context Directly to an Object

You can download a context directly in Python, modify it and then add it to the object you want to store. As an illustration we load a person context from json-ld.org (actually a fragment of the schema.org context) and insert it in a person object. (We have broken and truncated some output lines for clarity and all the code executed is repeated below for ease of copying.)

```

>>>
context=requests.get("https://json-ld.org/contexts/person.json
ld").json()['@context']
>>> context
{'Person': 'http://xmlns.com/foaf/0.1/Person',
 'xsd': 'http://www.w3.org/2001/XMLSchema#',
 'name': 'http://xmlns.com/foaf/0.1/name',
 'jobTitle': 'http://xmlns.com/foaf/0.1/title',
 'telephone': 'http://schema.org/telephone',
 'nickname': 'http://xmlns.com/foaf/0.1/nick',
 'affiliation': 'http://schema.org/affiliation',
 'depiction': {'@id': 'http://xmlns.com/foaf/0.1/depiction',
 '@type': '@id'},
 'image': {'@id': 'http://xmlns.com/foaf/0.1/img', '@type':
 '@id'},
 'born': {'@id': 'http://schema.org/birthDate', '@type':
 'xsd:date'},
 ...}
>>> person = {
    "@context": context,
    "@type": "Person",
    "@id": "foaf:person-1",
    "name": "Jane Doe",
    "jobTitle": "Professor",
    "telephone": "(425) 123-4567",
}
>>> test(person)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/name>, "Jane Doe")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/title>, "Professor")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://schema.org/telephone>, "(425) 123-4567")
(<http://xmlns.com/foaf/0.1/person-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://xmlns.com/foaf/0.1/Person>)
>>>

```

```

context=requests.get("https://json-ld.org/contexts/person.json
ld").json()['@context']
# The next produces lots of output, uncomment if desired

```

```
#context
```

```
person = {  
    "@context": context,  
    "@type": "Person",  
    "@id": "foaf:person-1",  
    "name": "Jane Doe",  
    "jobTitle": "Professor",  
    "telephone": "(425) 123-4567",  
}  
test(person)
```

Building a Graph of Objects

We start by forcing a key's value to be stored as a resource. We saw above that we could specify the value of a key to be a date using the `xsd:dateTime` specification. We now do it again for `foaf:birthdate`. Then we created several linked objects and show the connections using Gruff.

```
context = { "foaf:child": {"@type":"@id"},  
            "foaf:brotherOf": {"@type":"@id"},  
            "foaf:birthdate": {"@type":"xsd:dateTime"}}
```

```
p1 = {  
    "@context": context,  
    "@type": "foaf:Person",  
    "@id": "foaf:person-1",  
    "foaf:birthdate": "1958-04-09T20:00:00Z",  
    "foaf:child": ['foaf:person-2', 'foaf:person-3']  
}
```

```
p2 = {  
    "@context": context,  
    "@type": "foaf:Person",  
    "@id": "foaf:person-2",  
    "foaf:brotherOf": "foaf:person-3",  
    "foaf:birthdate": "1992-04-09T20:00:00Z",  
}
```

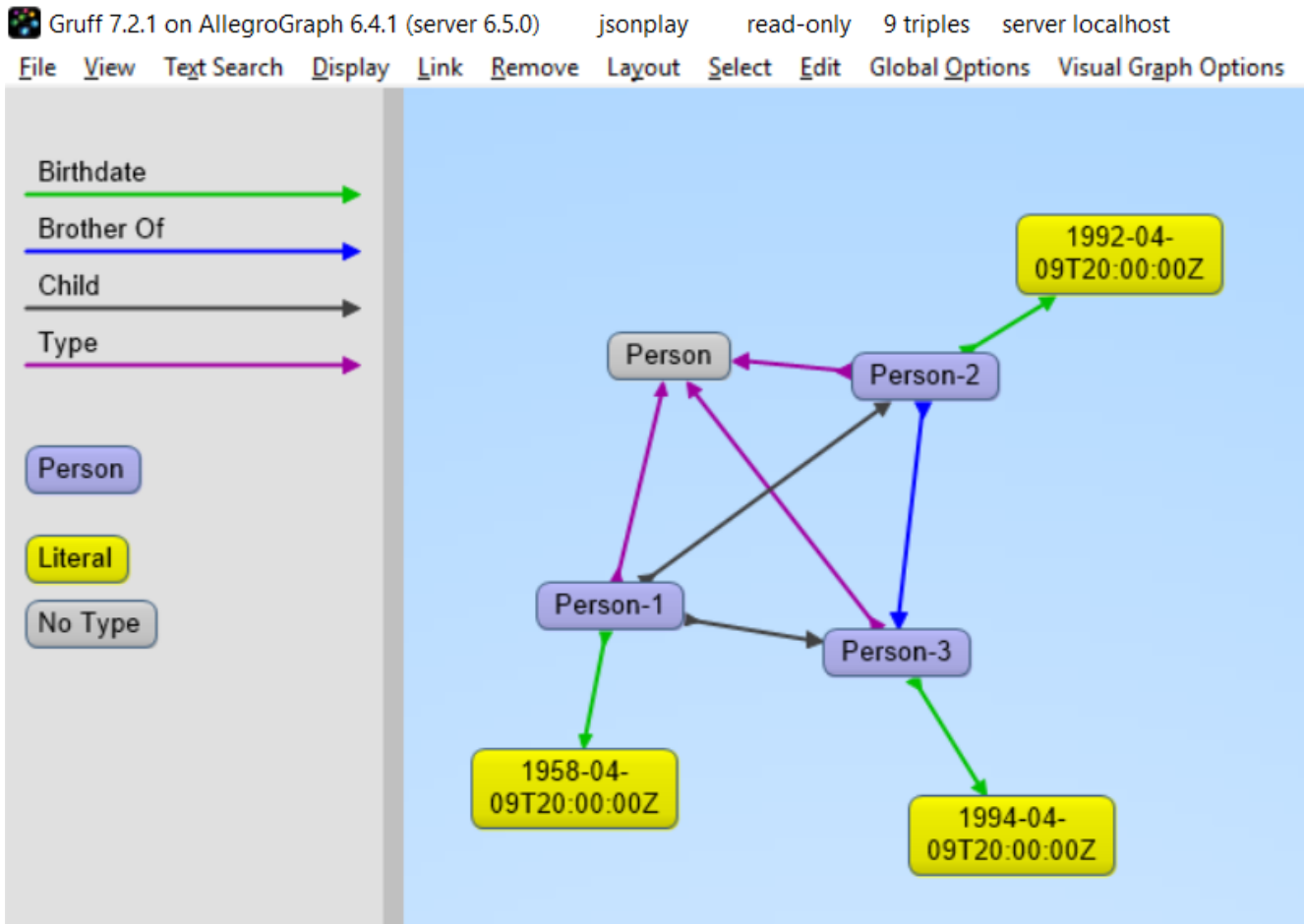
```
p3 = {"@context": context,
      "@type": "foaf:Person",
      "@id": "foaf:person-3",
      "foaf:birthdate": "1994-04-09T20:00:00Z",
}
```

```
test([p1,p2,p3])
```

```
>>> test([p1,p2,p3])
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/birthdate>,
"1958-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/child>,
<http://xmlns.com/foaf/0.1/person-2>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://xmlns.com/foaf/0.1/child>,
<http://xmlns.com/foaf/0.1/person-3>)
(<http://xmlns.com/foaf/0.1/person-1>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://xmlns.com/foaf/0.1/Person>)
(<http://xmlns.com/foaf/0.1/person-2>,
<http://xmlns.com/foaf/0.1/brotherOf>,
<http://xmlns.com/foaf/0.1/person-3>)
(<http://xmlns.com/foaf/0.1/person-2>,
<http://xmlns.com/foaf/0.1/birthdate>,
"1992-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://xmlns.com/foaf/0.1/person-2>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://xmlns.com/foaf/0.1/Person>)
(<http://xmlns.com/foaf/0.1/person-3>,
<http://xmlns.com/foaf/0.1/birthdate>,
"1994-04-09T20:00:00Z"^^<http://www.w3.org/2001/XMLSchema#date
Time>)
(<http://xmlns.com/foaf/0.1/person-3>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://xmlns.com/foaf/0.1/Person>)
```

The following shows the graph that we created in Gruff. Note

that this is what JSON-LD is all about: connecting objects together.



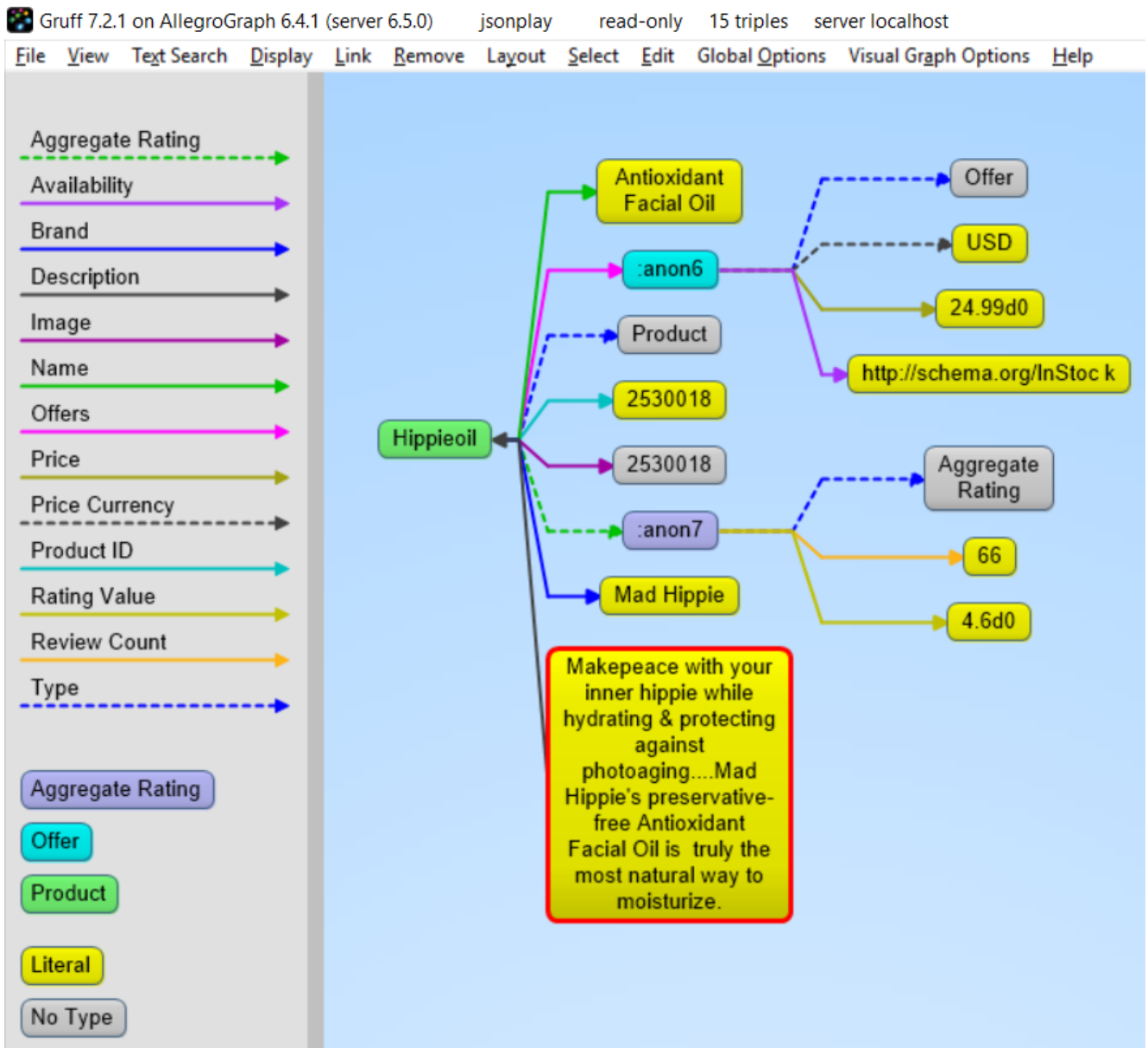
JSON-LD Keyword Directives can be Added at any Level

Here is an example from the wild. The URL <https://www.ulta.com/antioxidant-facial-oil?productId=xlsImpprod18731241> goes to a web page advertising a facial oil. (We make no claims or recommendations about this product. We are simply showing how JSON-LD appears in many places.) Look at the source of the page and you'll find a JSON-LD object similar to the following. Note that @ directives go to any level. We added an @id key.

```
hippieoil = {"@context":"http://schema.org",
  "@type":"Product",
  "@id":"http://franz.com/hippieoil",
```

```
"aggregateRating":
  {"@type": "AggregateRating",
    "ratingValue": 4.6,
    "reviewCount": 73},
  "description": "" "Make peace with your inner hippie while
hydrating & protecting against photoaging....Mad Hippie's
preservative-free Antioxidant Facial Oil is truly the most
natural way to moisturize.""",
  "brand": "Mad Hippie",
  "name": "Antioxidant Facial Oil",
  "image": "https://images.ulta.com/is/image/Ulta/2530018",
  "productID": "2530018",
  "offers":
    {"@type": "Offer",
      "availability": "http://schema.org/InStock",
      "price": "24.99",
      "priceCurrency": "USD"}}
```

```
test(hippieoil)
```



JSON-LD @graphs

One can put one or more JSON-LD objects in an RDF named graph. This means that the fourth element of each triple generated from a JSON-LD object will have the specified graph name. Let's show in an example.

```
context = {
  "name": "http://schema.org/name",
  "description": "http://schema.org/description",
  "image": {
    "@id": "http://schema.org/image", "@type": "@id"
  },
}
```

```

        "geo": "http://schema.org/geo",
        "latitude": {
            "@id": "http://schema.org/latitude", "@type":
"xsd:float" },
        "longitude": {
            "@id": "http://schema.org/longitude", "@type":
"xsd:float" },
        "xsd": "http://www.w3.org/2001/XMLSchema#"
    }

```

```

place = {
    "@context": context,
    "@id": "http://franz.com/place1",
    "@graph": {
        "@id": "http://franz.com/place1",
        "@type": "http://franz.com/Place",
        "name": "The Empire State Building",
        "description": "The Empire State Building is a 102-
story landmark in New York City.",
        "image":
"http://www.civil.usherbrooke.ca/cours/gci215a/empire-state-bu
ilding.jpg",
        "geo": {
            "latitude": "40.75",
            "longitude": "73.98" }
    }}

```

and here is the result:

```

>>> test(place, maxPrint=3)
(<http://franz.com/place1>, <http://schema.org/name>, "The
Empire State Building", <http://franz.com/place1>)
(<http://franz.com/place1>, <http://schema.org/description>,
"The Empire State Building is a 102-story landmark in New York
City.", <http://franz.com/place1>)
(<http://franz.com/place1>, <http://schema.org/image>,
<http://www.civil.usherbrooke.ca/cours/gci215a/empire-state-bu
ilding.jpg>, <http://franz.com/place1>)
>>>

```

Note that the fourth element (graph) of each of the triples is

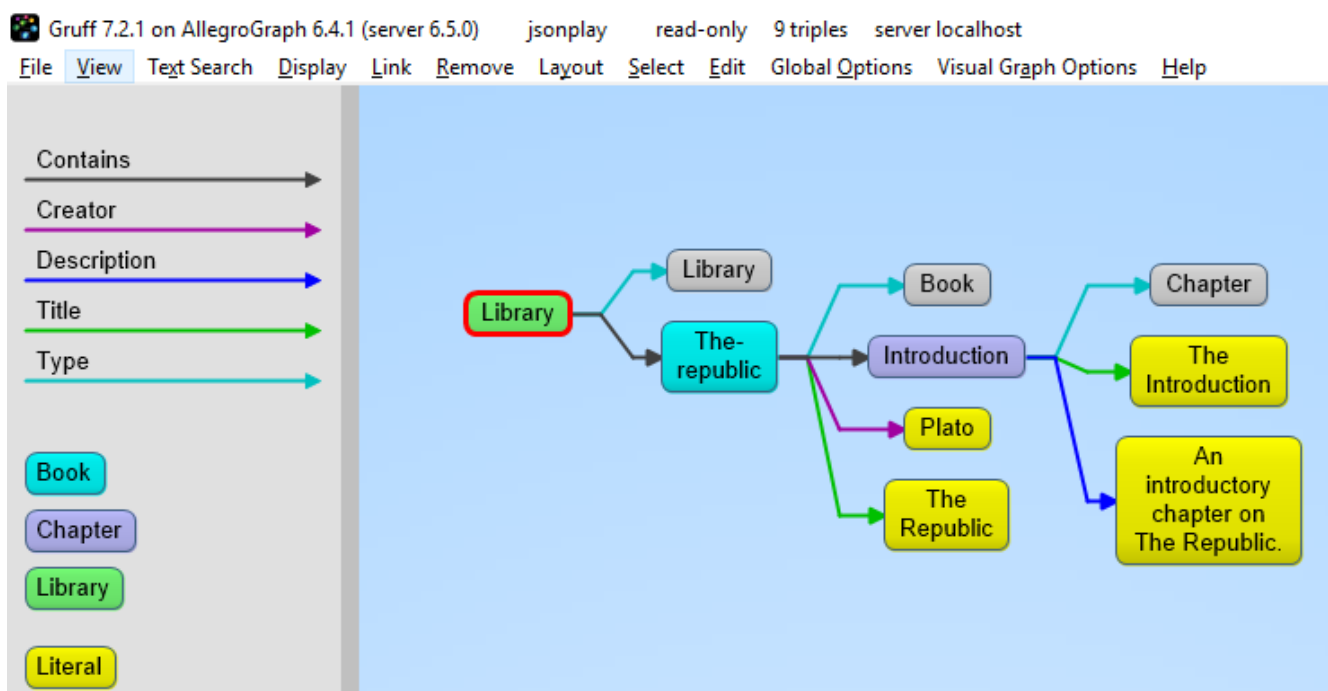
<http://franz.com/placel>. If you don't add the @id the triples will be put in the default graph.

Here a slightly more complex example:

```
library = {
  "@context": {
    "dc": "http://purl.org/dc/elements/1.1/",
    "ex": "http://example.org/vocab#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ex:contains": {
      "@type": "@id"
    }
  },
  "@id": "http://franz.com/mygraph1",
  "@graph": [
    {
      "@id": "http://example.org/library",
      "@type": "ex:Library",
      "ex:contains": "http://example.org/library/the-republic"
    },
    {
      "@id": "http://example.org/library/the-republic",
      "@type": "ex:Book",
      "dc:creator": "Plato",
      "dc:title": "The Republic",
      "ex:contains":
"http://example.org/library/the-republic#introduction"
    },
    {
      "@id":
"http://example.org/library/the-republic#introduction",
      "@type": "ex:Chapter",
      "dc:description": "An introductory chapter on The
Republic.",
      "dc:title": "The Introduction"
    }
  ]
}
```

With the result:

```
>>> test(library, maxPrint=3)
(<http://example.org/library>,
<http://example.org/vocab#contains>,
<http://example.org/library/the-republic>,
<http://franz.com/mygraph1>) (<http://example.org/library>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://example.org/vocab#Library>,
<http://franz.com/mygraph1>)
(<http://example.org/library/the-republic>,
<http://purl.org/dc/elements/1.1/creator>,
"Plato", <http://franz.com/mygraph1>)
>>>
```



JSON-LD as a Document Store

So far we have treated JSON-LD as a syntax to create triples. Now let us look at the way we can start using AllegroGraph as a combination of a document store and graph database at the same time. And also keep in mind that we want to do it in such a way that you as a Python developer can add documents such as dictionaries and also retrieve values or documents as dictionaries.

Setup

The Python source file `jsonld_tutorial_helper.py` contains various definitions useful for the remainder of this example. Once it is downloaded, do the following (after adding the path to the filename):

```
conn=createdb("docugraph")
from jsonld_tutorial_helper import *
addNamespace(conn,"jsonldmeta","http://franz.com/ns/allegrograph/6.4/load-meta#")
addNamespace(conn,"ical","http://www.w3.org/2002/12/cal/ical#"
)
```

Let's use our event structure again and see how we can store this JSON document in the store as a document. Note that the `addData` call includes the keyword: `json_ld_store_source=True`.

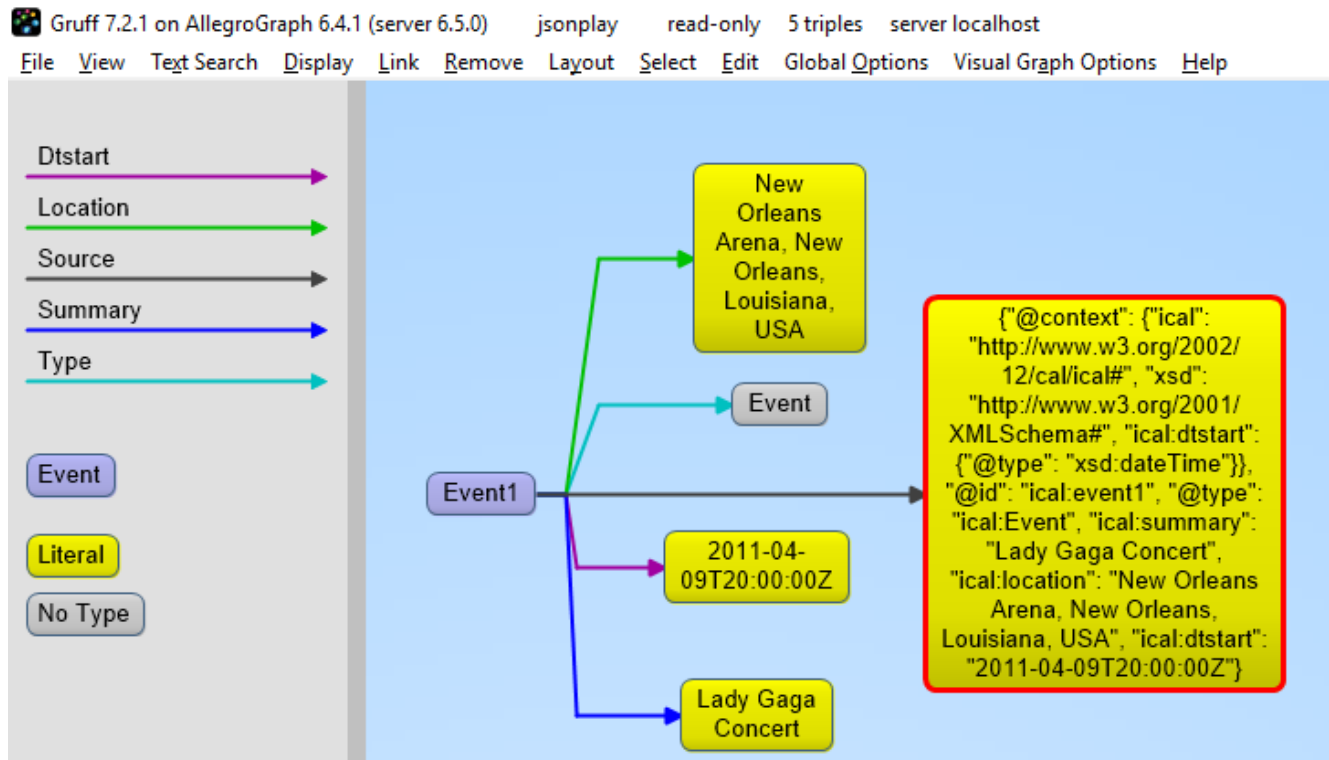
```
event = {
    "@context": {
        "@id": "ical:event1",
        "@type": "ical:Event",
        "ical": "http://www.w3.org/2002/12/cal/ical#",
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "ical:dtstart": { "@type": "xsd:dateTime" }
    },
    "ical:summary": "Lady Gaga Concert",
    "ical:location":
    "New Orleans Arena, New Orleans, Louisiana, USA",
    "ical:dtstart": "2011-04-09T20:00:00Z"
}
```

```
>>> conn.addData(event,
allow_external_references=True,json_ld_store_source=True)
```

The `jsonld_tutorial_helper.py` file defines the function `store` as simple wrapper around `addData` that always saves the JSON source. For experimentation reasons it also has a parameter `fresh` to clear out the repository first.

```
>>> store(conn,event, fresh=True)
```

If we look at the triples in Gruff we see that the JSON source is stored as well, on the root (top-level *@id*) of the JSON object.



For the following part of the tutorial we want a little bit more data in our repository so please look at the helper file *jsonld_tutorial_helper.py* where you will see that at the end we have a dictionary named *obs* with about 9 diverse objects, mostly borrowed from the json-ld.org site: a person, an event, a place, a recipe, a group of persons, a product, and our hippieoil.

First let us store all the objects in a fresh repository. Then we check the size of the repo. Finally, we create a freetext index for the JSON sources.

```
>>> store(conn,[v for k,v in obs.items()], fresh=True)
>>> conn.size()
86
>>>
conn.createFreeTextIndex("source",['<http://franz.com/ns/alleg
```

```
rograph/6.4/load-meta#source>']])  
>>>
```

Retrieving values with SPARQL

To simply retrieve values in objects but not the objects themselves, regular SPARQL queries will suffice. But because we want to make sure that Python developers only need to deal with regular Python structures as lists and dictionaries, we created a simple wrapper around SPARQL (see helper file). The name of the wrapper is `runSparql`.

Here is an example. Let us find all the roots (top-level *@ids*) of objects and their types. Some objects do not have roots, so `None` stands for a blank node.

```
>>> pprint(runSparql(conn,"select ?s ?type { ?s a ?type }"))  
[{'s': 'cocktail1', 'type': 'Cocktail'},  
 {'s': None, 'type': 'Individual'},  
 {'s': None, 'type': 'Vehicle'},  
 {'s': 'tesla', 'type': 'Offering'},  
 {'s': 'place1', 'type': 'Place'},  
 {'s': None, 'type': 'Offer'},  
 {'s': None, 'type': 'AggregateRating'},  
 {'s': 'hippieoil', 'type': 'Product'},  
 {'s': 'person-3', 'type': 'Person'},  
 {'s': 'person-2', 'type': 'Person'},  
 {'s': 'person-1', 'type': 'Person'},  
 {'s': 'person-1000', 'type': 'Person'},  
 {'s': 'event1', 'type': 'Event'}]  
>>>
```

We do not see the full URIs for `?s` and `?type`. You can see them by adding an appropriate *format* argument to `runSparql`, but the default is `terse`.

```
>>> pprint(runSparql(conn,"select ?s ?type { ?s a ?type }  
limit 2",format='ntriples'))  
[{'s': '<http://franz.com/cocktail1>', 'type':
```

```
'<http://franz.com/Cocktail>'},
      {'s': None, 'type':
'<http://purl.org/goodrelations/v1#Individual>'}]}
>>>
```

Retrieving a Dictionary or Object

`retrieve` is another function defined (in *jsonld_tutorial_helper.py*) for this tutorial. It is a wrapper around SPARQL to help extract objects. Here we see how we can use it. The sole purpose of `retrieve` is to retrieve the JSON-LD/dictionary based on a SPARQL pattern.

```
>>> retrieve(conn,"{?this a ical:Event}")
[{'@type': 'ical:Event', 'ical:location': 'New Orleans Arena,
New Orleans, Louisiana, USA', 'ical:summary': 'Lady Gaga
Concert', '@id': 'ical:event1', '@context': {'xsd':
'http://www.w3.org/2001/XMLSchema#', 'ical':
'http://www.w3.org/2002/12/cal/ical#', 'ical:dtstart':
{'@type': 'xsd:dateTime'}}, 'ical:dtstart':
'2011-04-09T20:00:00Z'}]
>>>
```

Ok, for a final fun (if you like expensive cars) example: Let us find a thing that is “fast and furious”, that is worth more than \$80,000 and that we can pay for in cash:

```
>>>
addNamespace(conn,"gr","http://purl.org/goodrelations/v1#")
>>> x = retrieve(conn, """"{ ?this fti:match 'fast furious*';
      gr:acceptedPaymentMethods gr:Cash ;
      gr:hasPriceSpecification ?price .
      ?price gr:hasCurrencyValue ?value ;
      gr:hasCurrency "USD" .
      filter ( ?value > 80000.0 ) }""")
>>> pprint(x)
[{'@context': {'foaf': 'http://xmlns.com/foaf/0.1/',
'foaf:page': {'@type': '@id'},
'gr': 'http://purl.org/goodrelations/v1#',
```

```

        'gr:acceptedPaymentMethods': {'@type': '@id'},
        'gr:hasBusinessFunction': {'@type': '@id'},
        'gr:hasCurrencyValue': {'@type': 'xsd:float'},
        'pto': 'http://www.productontology.org/id/',
        'xsd': 'http://www.w3.org/2001/XMLSchema#'},
    '@id': 'http://example.org/cars/for-sale#tesla',
    '@type': 'gr:Offering',
    'gr:acceptedPaymentMethods': 'gr:Cash',
    'gr:description': 'Need to sell fast and furiously',
    'gr:hasBusinessFunction': 'gr:Sell',
    'gr:hasPriceSpecification': {'gr:hasCurrency': 'USD',
                                'gr:hasCurrencyValue':
'85000'},
    'gr:includes': {'@type': ['gr:Individual', 'pto:Vehicle'],
                    'foaf:page':
'http://www.teslamotors.com/roadster',
                    'gr:name': 'Tesla Roadster'},
    'gr:name': 'Used Tesla Roadster'}}
>>> x[0]['@id']
'http://example.org/cars/for-sale#tesla'

```

ГРАФОВЫЕ БАЗЫ: ПРИНЦИП РАБОТЫ И ПРИМЕНЕНИЕ — GRAPH BASES: PRINCIPLE OF OPERATION AND APPLICATION

Всеволод Дёмкин удаленно работает во Franz Inc. над графовой базой AllegroGraph. Преподает в Projector курс «Natural Language Processing». В свободное время делает open-сорс для обработки природных текстов на Lisp'е.

Мы рассмотрим создание программы для агрегации текстов из разных источников, таких как twitter, блоги, reddit и т.д., —

их автоматической, а затем ручной обработки для формирования дайджеста новостей по определенной теме. На этом примере мы проанализируем, какие преимущества дает использование графовых баз данных, обсудим их возможности и ограничения.

В качестве конкретной БД будет использована система Franz AllegroGraph и мы ознакомимся с ее экосистемой, включающей возможности построение API и веб-приложений, а также со средой Allegro Common Lisp, на которой она построена. Особое внимание будет уделено использованию машинного обучения и NLP при решении задач работы с текстом, в частности, внутри AllegroGraph.

Обсудим:

- В чем особенности, как работают, преимущества/недостатки графовых БД;
- Как решать базовые задачи обработки текстов с использованием инструментария ML/NLP;
- Как построить полноценное приложение с ядром обработки текста на основе графовой БД и ML/NLP технологий;
- Как устроена экосистема Common Lisp и как можно задействовать ее для создания серверных приложений.

Лекция будет полезна: разработчикам, которые интересуются темой графовых баз данных и/или ML/NLP.

What is the most interesting

use of a graph database you ever seen? PWC responds.

From a Quora post by Alan Morrison – Sr. Research Fellow at PricewaterhouseCoopers – November 2018

The most interesting use is the most powerful: standard RDF graphs for large-scale knowledge graph integration.

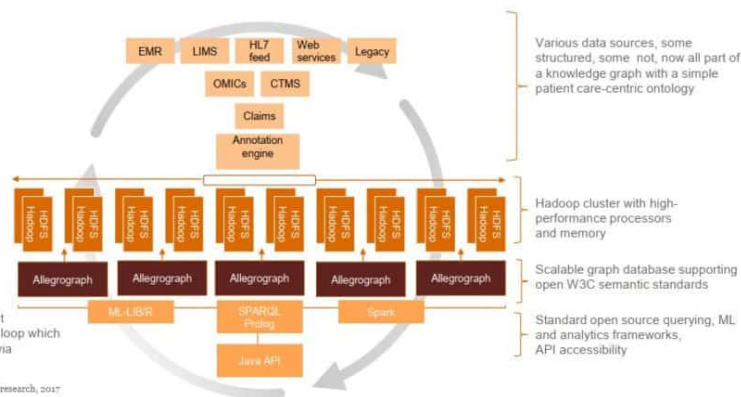
*From my notes on a talk Parsa Mirhaji of Montefiore Health System gave in 2017. **Montefiore uses Franz AllegroGraph, a distributed RDF graph database.** He modeled a core patient-centric hospital knowledge need using a simple standard ontology with a 1,000 or so concepts total.*

That model integrated data from lots of different kinds of heterogeneous sources so that doctors could query the knowledge graph from tablets or phones at a patient's bedside and get contextualized, patient-specific answers to questions for diagnostic purposes.

Fast forward to 2018, and nine out of ten of the most value-creating companies in the world are using standard knowledge graphs in a comparable fashion, either as a base for multi-domain intelligent assistants a la Siri or Alibot or Alexa, or to integrate and contextualize business domains cross-enterprise, or both. The method is preparatory to what John Launchbury of DARPA described as the Third Wave of AI.....

Montefiore's semantic data lake

Doctors can query the graph or harness ML + analytics and receive answers from the system at the point of care via their handhelds.



Montefiore Health, Franz, Intel and PwC research, 2017

PwC | Collapsing the IT stack

29

Read the full article over at Quora

2019 Trends in Data Governance: The Model Governance Question

From an AI Business Article by Jelani Harper – November 2018

The propagation of the enterprise's ability to capitalize on data-driven processes—to effectively reap data's yield as an organizational asset, much like any other—hinges on data governance, which arguably underpins the foundation of data management itself.

There are numerous trends impacting that foundation, many of which have always had, and will continue to have, relevance as 2019 looms. Questions of regulatory compliance, data

lineage, metadata management, and even data governance will all play crucial roles.

Franz's CEO, Dr. Jans Aasman was quoted:

Still, as Aasman denoted, "It's extremely complicated to make fair [machine learning] models with all the context around them." Both rules and human supervision of models can furnish a fair amount of context for them, serving as starting points for their consistent governance.

Read the full article at [AI Business](#).

AI Requires More Than Machine Learning

From Forbes Technology Council – October 2018

This article discusses the facets of machine learning and AI:

Lauded primarily for its automation and decision support, machine learning is undoubtedly a vital component of artificial intelligence. However, a small but growing number of thought leaders throughout the industry are acknowledging that the breadth of AI's upper cognitive capabilities involves more than just machine learning.

Machine learning is all about sophisticated pattern recognition. It's virtually unsurpassable at determining relevant, predictive outputs from a series of data-driven inputs. Nevertheless, there is a plethora of everyday, practical business problems that cannot be solved with input/output reasoning alone. The problems also require the

multistep, symbolic reasoning of rules-based systems.

Whereas machine learning is rooted in a statistical approach, symbolic reasoning is predicated on the symbolic representation of a problem usually rooted in a knowledge base. Most rules-based systems involve multistep reasoning, including those powered by coding languages such as Prolog.

Read the full article over at Forbes

.

Transmuting Machine Learning into Verifiable Knowledge

From AI Business – August 2018

This article covers machine learning and AI:

According to Franz CEO Jans Aasman, these machine learning deployments not only maximize organizational investments in them by driving business value, but also optimize the most prominent aspects of the data systems supporting them.

“You start with the raw data...do analytics on it, get interesting results, then you put the results of the machine learning back in the database, and suddenly you have a far more powerful database,” Aasman said.

Dr. Aasman is further quoted:

For internal applications, organizations can use machine learning concepts (such as co-occurrence—how often defined concepts occur together) alongside other analytics to monitor employee behavior, efficiency, and success with customers or certain types of customers. Aasman mentioned a project management use case for a consultancy company in which these analytics were used to “compute for every person, or every combination of persons, whether or not the project was successful: meaning, done on time to the satisfaction of the customer.”

Organizations can use whichever metrics are relevant for their businesses to qualify success. This approach is useful for determining a numerical rating for employees “and you could put that rating back in the database,” Aasman said. “Now you can do a follow up query where you say how much money did I make on the top 10 successful people; how much money did I lose on the top 10 people I don’t make a profit on.”

Read the full article over at [AI Business](#).