Webcast – Speech Recognition, Knowledge Graphs, and AI for Intelligent Customer Operations – April 3, 2019

Presenters – Burt Smith, N3 Results and Jans Aasman, Franz Inc.

In the typical sales organization the contents of the actual chat or voice conversation between agent and customer is a black hole. In the modern Intelligent Customer Operations center (e.g. N3 Results – www.n3results.com) the interactions between agent and customer are a source of rich information that helps agents to improve the quality of the interaction in real time, creates more sales, and provides far better analytics for management.

Join us for this Webinar where we describe a real world Intelligent Customer Operations center that uses graph based technology for taxonomy driven entity extraction, speech recognition, machine learning and predictive analytics to improve quality of conversations, increase sales and improve business visibility.

View the recorded webinar.

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='xyzzy') 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="xyzzy",create=True,clear=True)

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

than 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.jsonld").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

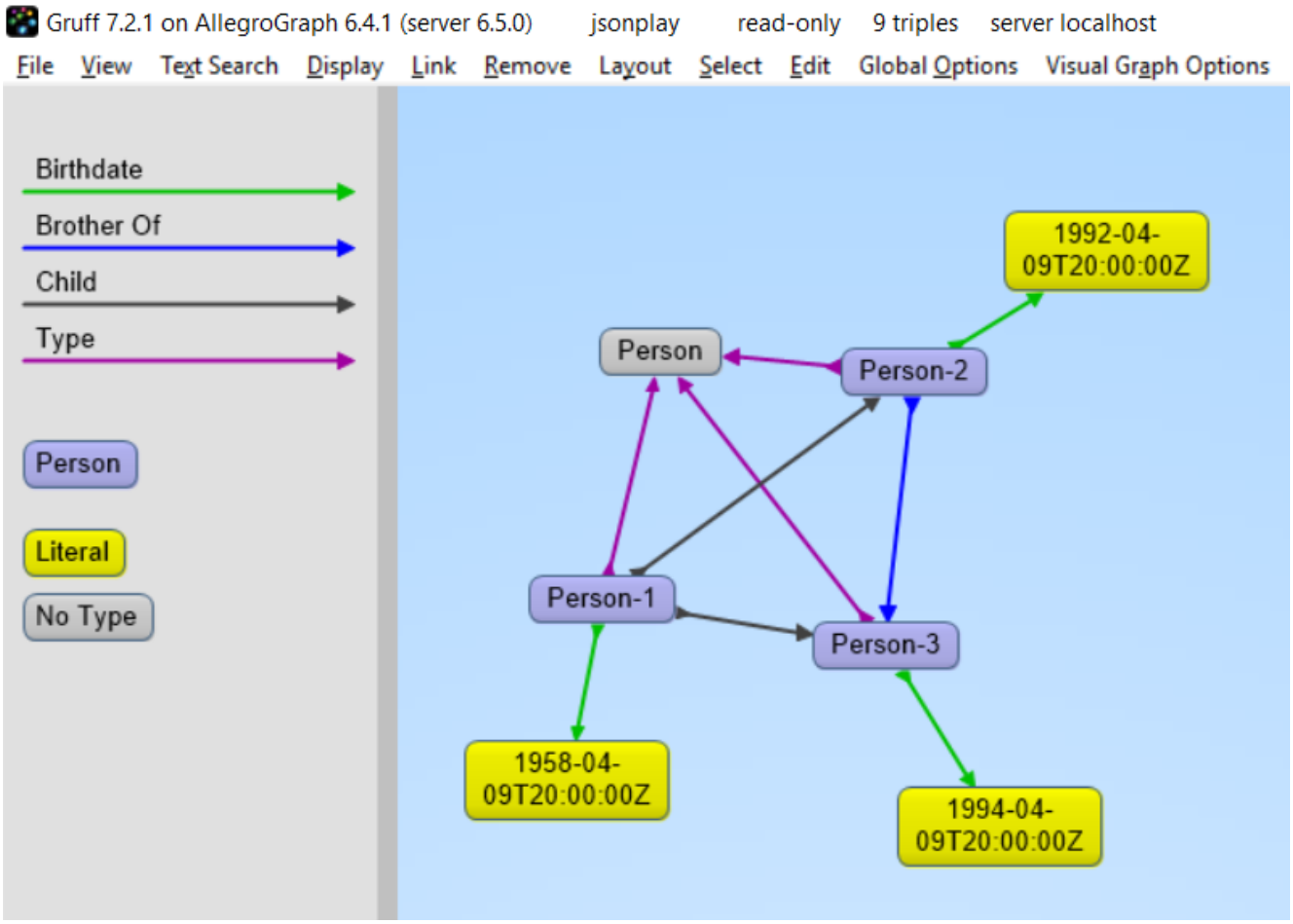
We start by forcing a key's value to be stored as a resource. We saw above that we could specify the value of a key to be a date using the xsd:dateTime specification. We now do it again for foaf:birthdate. Then we created several linked objects and show the connections using Gruff.

```
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.



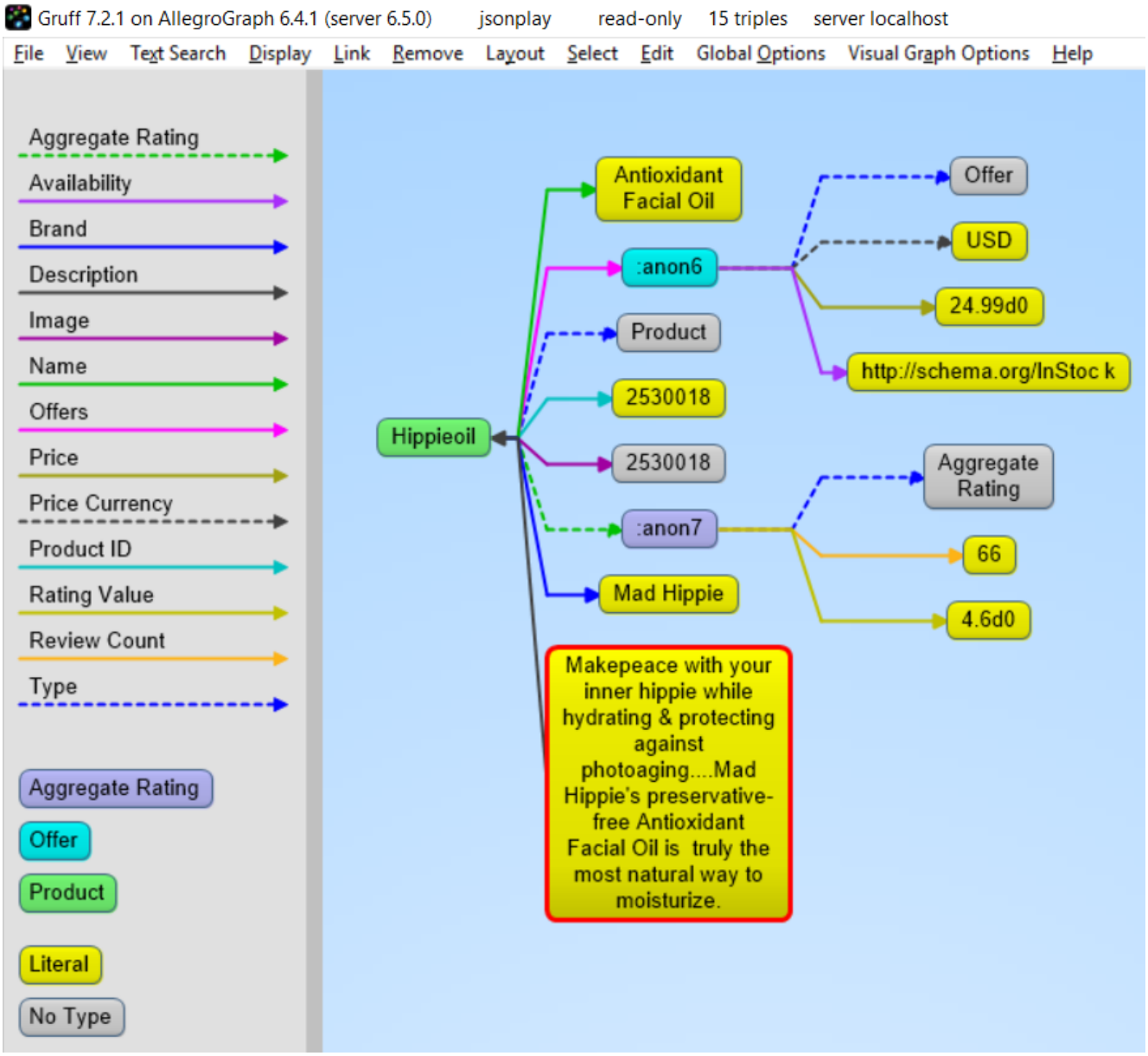
JSON-LD Keyword Directives can be Added at any Level

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

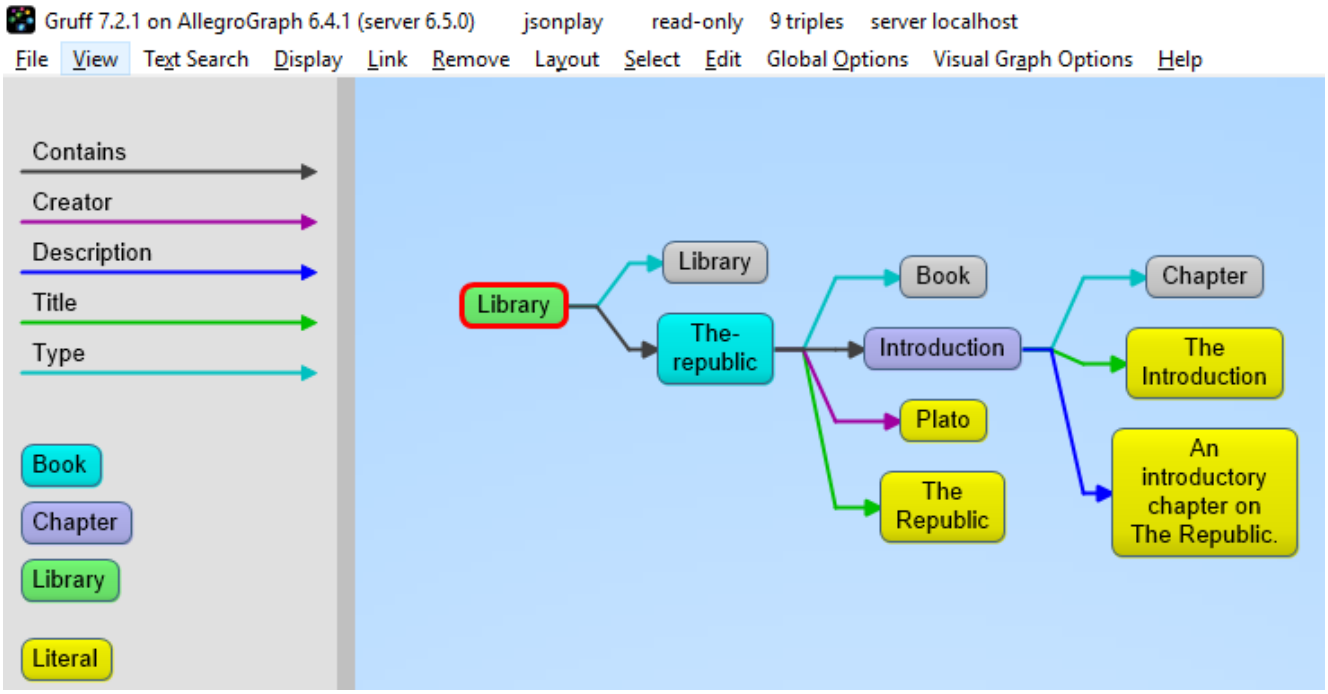
<http://franz.com/place1>. If you don’t add the @id the triples will be put in the default graph.

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.

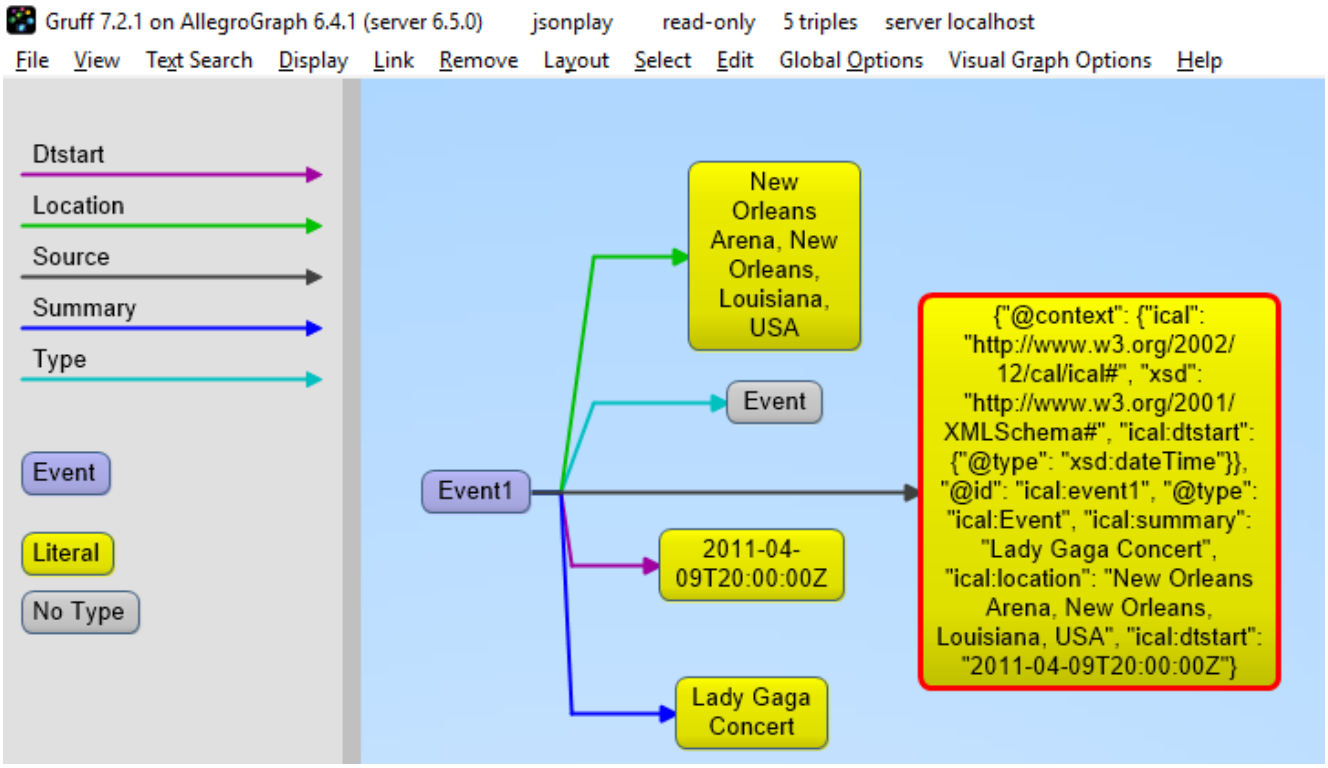
```
event = {
  "@context": {
    "@id": "ical:event1",
    "@type": "ical:Event",
    "ical": "http://www.w3.org/2002/12/cal/ical#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ical:dtstart": { "@type": "xsd:dateTime" }
      },
    "ical:summary": "Lady Gaga Concert",
    "ical:location":
    "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 addDatathat 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'
```

Gartner Identifies Top 10 Data and Analytics Technology Trends for 2019

According to Donald Feinberg, vice president and distinguished analyst at Gartner, the very challenge created by digital disruption – too much data – has also created an unprecedented opportunity. The vast amount of data, together with increasingly powerful processing capabilities enabled by the cloud, means it is now possible to train and execute algorithms at the large scale necessary to finally realize the full potential of AI.

“The size, complexity, distributed nature of data, speed of action and the continuous intelligence required by digital business means that rigid and centralized architectures and tools break down,” Mr. Feinberg said. “The continued survival of any business will depend upon an agile, data-centric architecture that responds to the constant rate of change.”

Gartner recommends that data and analytics leaders talk with senior business leaders about their critical business priorities and explore how the following top trends can enable them.

Trend No. 5: Graph

Graph analytics is a set of analytic techniques that allows for the exploration of relationships between entities of interest such as organizations, people and transactions.

The application of graph processing and graph DBMSs will grow at 100 percent annually through 2022 to continuously accelerate data preparation and enable more complex and adaptive data science.

Graph data stores can efficiently model, explore and query data with complex interrelationships across data silos, but the need for specialized skills has limited their adoption to date, according to Gartner.

Graph analytics will grow in the next few years due to the need to ask complex questions across complex data, which is not always practical or even possible at scale using SQL queries.

https://www.gartner.com/en/newsroom/press-releases/2019-02-18-gartner-identifies-top-10-data-and-analytics-technolo

What is the Answer to AI Model Risk Management?

Algorithm-XLab – March 2019

Franz CEO Dr. Jans Aasman Explains how to manage AI Modelling Risks.

AI model risk management has moved to the forefront of contemporary concerns for statistical Artificial Intelligence, perhaps even displacing the notion of ethics in this regard because of the immediate, undesirable repercussions of tenuous machine learning and deep learning models.

AI model risk management requires taking steps to ensure that the models used in artificial applications produce results that are unbiased, equitable, and repeatable.

The objective is to ensure that given the same inputs, they produce the same outputs.

If organizations cannot prove how they got the results of AI risk models, or have results that are discriminatory, they are subject to regulatory scrutiny and penalties.

Strict regulations throughout the financial services industry in the United Statesand Europe require governing, validating, re-validating, and demonstrating the transparency of models

for financial products.

There’s a growing cry for these standards in other heavily regulated industries such as healthcare, while the burgeoning Fair, Accountable, Transparent movementtypifies the horizontal demand to account for machine learning models’ results.

AI model risk management is particularly critical in finance.

Financial organizations must be able to demonstrate how they derived the offering of any financial product or service for specific customers.

When deploying AI risk models for these purposes, they must ensure they can explain (to customers and regulators) the results that determined those offers.

Read the full article at Algorithm-XLab.

New!!! AllegroGraph v6.5 – Multi-model Semantic Graph and Document Database

Download – **AllegroGraph v6.5 and Gruff v7.3**

AllegroGraph – **Documentation**

Gruff – **Documentation**

Adding JSON/JSON-LD Documents to a Graph Database

Traditional document databases (e.g. MongoDB) have excelled at storing documents at scale, but are not designed for linking

data to other documents in the same database or in different databases. AllegroGraph 6.5 delivers the unique power to define many different types of documents that can all point to each other using standards-based semantic linking and then run SPARQL queries, conduct graph searches, execute complex joins and even apply Prolog AI rules directly on a diverse sea of objects.

AllegroGraph 6.5 provides free text indexes of JSON documents for retrieval of information about entities, similar to document databases. But unlike document databases, which only link data objects within documents in a single database, AllegroGraph 6.5 moves the needle forward in data analytics by semantically linking data objects across multiple JSON document stores, RDF databases and CSV files. Users can run a single SPARQL query that results in a combination of structured data and unstructured information inside documents and CSV files. AllegroGraph 6.5 also enables retrieval of entire documents.

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

A direct benefit for companies using AllegroGraph is that they now can combine their documents with graphs, graph search and graph algorithms. Normally when you store documents in a document database you set up your documents in such a way that it is optimized for certain direct retrieval queries. Performing complex joins for multiple types of documents or even performing a shortest path through a mass of object (types) is too complicated. Storing JSON-LD objects in AllegroGraph gives users all the benefits of a document database AND the ability to semantically link objects together, run complex joins, and perform graph search queries.

Another key benefit for companies is that your application developers don’t have to learn the entire semantic technology stack, especially the part where developers have to create individual RDF triples or edges. Application developers love to work with JSON data as serialization for objects. In JavaScript the JSON format is syntactically identical to the code for creating JavaScript objects and in Python the most import data structure is the ‘dictionary’ which is also near identical to JSON.

Key AllegroGraph v6.5 Features:

- Support for loading JSON-LD and also some non-RDF data files, that is files which are not already organized into triples or quads. See Loading non-RDF data section in the Data Loading document for more information on loading non-RDF data files. Loading JSON-LD files is described along with other RDF formats in the Data Loading document. The section Supported RDF formats lists all supported RDF formats.

- Support for two phase commits (2PC), which allows AllegroGraph to participate in distributed transactions compromising a number of AllegroGraph and non-AllegroGraph databases (e.g. MongoDB, Solr, etc), and to ensure that the work of a transaction must either be committed on all participants or be rolled back on all participants. Two-phase commit is described in the Two-phase commit document.

- An event scheduler: Users can schedule events in the future. The event specifies a script to run. It can run once or repeatedly on a regular schedule. See the Event Scheduler document for more information.

- AllegroGraph is 100 percent ACID, supporting Transactions: Commit, Rollback, and Checkpointing. Full and Fast Recoverability. Multi-Master Replication
- Triple Attributes – Quads/Triples can now have attributes which can provide fine access control.
- Data Science – Anaconda, R Studio
- 3D and multi-dimensional geospatial functionality
- SPARQL v1.1 Support for Geospatial, Temporal, Social Networking Analytics, Hetero Federations
- Cloudera, Solr, and MongoDB integration
- JavaScript stored procedures
- RDF4J Friendly, Java Connection Pooling
- Graphical Query Builder for SPARQL and Prolog – Gruff
- SHACL (Beta) and SPIN Support (SPARQL Inferencing Notation)
- AGWebView – Visual Graph Search, Query Interface, and DB Management
- Transactional Duplicate triple/quad deletion and suppression
- Advanced Auditing Support
- Dynamic RDFS++ Reasoning and OWL2 RL Materializer
- AGLoad with Parallel loader optimized for both traditional spinning media and SSDs.

Numerous other optimizations, features, and enhancements. Read the release notes – https://franz.com/agraph/support/documentation/current/release-notes.html