Big Data 50 – Companies Driving Innovation in 2019

Franz Inc. is proud to announce that it has been named to Database Trends and Application (DBTA) – Big Data 50, Companies Driving Innovation in 2019

Today, more than ever, businesses rely on data to deliver a competitive edge. The urgency to compete on analytics has spread across industries, fueled by the need for greater efficiency, agility and innovation," remarked Thomas Hogan, Group Publisher at Database Trends and Applications. "This list seeks to highlight those companies that are really driving innovation and serve as a guide to businesses navigating the rapidly changing big data landscape."

A new generation of tools is making it possible to leverage the wealth of data flowing into organizations from a previously unimaginable range of data sources. Machine learning, AI, Spark, and object storage are just some of the next-generation approaches gaining traction, according to recent surveys conducted by Unisphere Research, a division of Information Today, Inc.

But, it is also increasingly clear that there is no single way to approach data-driven innovation today. Open source-based technologies have gained strong adoption in organizations alongside proprietary offerings, data lakes are increasingly

being implemented but data warehouses continue in widespread use, and hybrid deployments spanning cloud and on-premise are commonly accepted.

Organizations are seeking to use data-driven innovation for better reporting and analytics, real-time decision making, enhanced customer experience and personalization, and reduced costs. But with data coming in from more places than ever, being stored in more systems, and accessed by more users for a wider array of use cases, there is greater recognition that security and governance must be addressed intelligently.

Evaluating new and disruptive technologies, and then identifying how and where they can be useful, can be challenging.

To contribute to the discussion each year, Big Data Quarterly presents the "Big Data 50," a list of forward-thinking companies that are working to expand what's possible in terms of capturing, storing, protecting, and deriving value from data.

"We are honored to receive this acknowledgement for our efforts in delivering Enterprise Knowledge Graph Solutions," said Dr. Jans Aasman, CEO, Franz Inc. "In the past year, we have seen demand for Enterprise Knowledge Graphs take off across industries along with recognition from top technology analyst firms that Knowledge Graphs provide the critical foundation for artificial intelligence applications and predictive analytics. Our AllegroGraph Knowledge Graph Platform Solution offers a unique comprehensive approach for helping companies accelerate the creation of Enterprise Knowledge Graphs that deliver new value to their organization."

Harnessing the Internet of Things with JSON-LD

Franz's CEO, Jans Aasman's recent IoT Evolution Article:

Conceptually, the promise of the Internet of Things is almost halcyon. Its billions of sensors are all connected, continuously transmitting data to support tailored, cost-saving measures maximizing revenues in applications as diverse as smart cities, smart price tags, and predictive maintenance in the Industrial Internet.

Practically, the data management necessities of capitalizing on this promise by the outset of the next decade are daunting. The vast majority of these datasets are unstructured or semi-structured. The data modeling challenges of rectifying their schema for integration are considerable. The low latency action required to benefit from their data implies machine intelligence largely elusive to today's organizations.

……

The self-describing, linked data approach upon which JSON-LD is founded excels at the low latent action resulting from machine to machine communication in the IoT. The nucleus of the linked data methodology—semantic statements and their unique Uniform Resource Identifiers (URIs)—are read and understood by machines. This characteristic aids many of the IoT use cases requiring machine intelligence; by transmitting

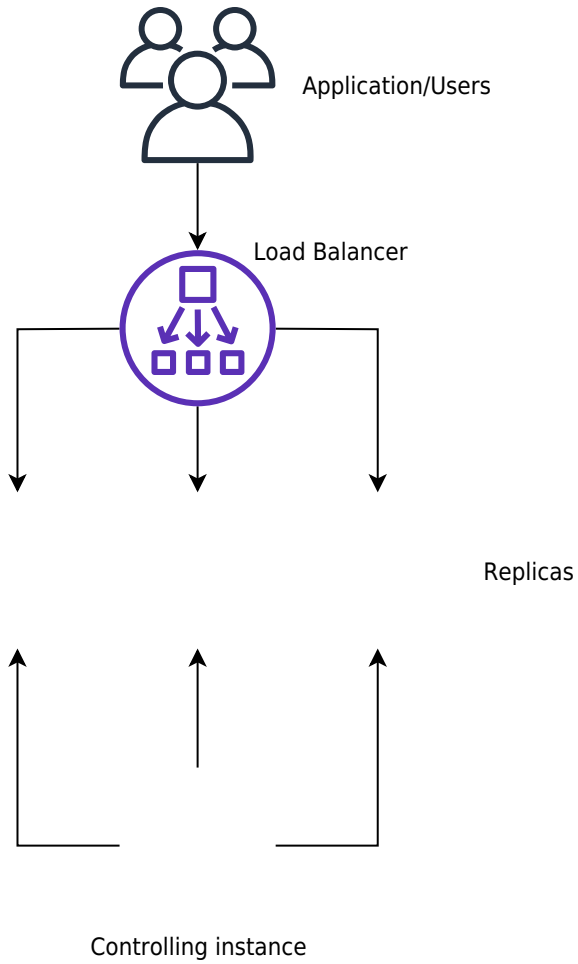
IoT data via the JSON-LD format organizations can maximize this boon. Smart cities provide particularly compelling examples of the machine intelligence fortified by this expression of semantic technology.

Read the full article at IoT Evolution

AllegroGraph Replication on Amazon's AWS using Terraform

Introduction

In this document we describe how to setup an AllegroGraph replication cluster on AWS using the terraform program. The cluster will have one controlling instance and a set of instances controlled by an Auto Scaling Group and reached via a Load Balancer.



Creating such a system on AWS takes a long time if done manually through their web interface. We have another document that takes you through the steps. Describing the system in terraform first takes a little time but once that's done the cluster can be started in less than five minutes.

Steps

1. Obtain an AMI with AllegroGraph and aws-repl (our support code for aws) installed.
2. Edit the terraform file we supply to suit your needs
3. Run terraform to build the cluster

Obtain an AMI with AllegroGraph and aws-repl

An AMI is an image of a virtual machine. You create an AMI by launching an ec2 instance using an AMI, altering the root disk of that instance and then telling AWS to create an AMI based on your instance. You can repeat this process until you create the AMI you need.

We have a prebuild AMI with all the code installed. It uses AllegroGraph 6.5.0 and doesn’t contain a license code so it’s limited to 5 million triples. You can use this AMI to test the load balancer or you can use this image as the starting off point for building your own image.

Alternatively you start from a fresh AMI and install everything yourself as described next.

We will create an AMI to run AllegroGraph with Replication with the following features

1. When an EC2 instance running this AMI is started it starts AllegroGraph and joins the cluster of nodes serving a particular repository.
2. When the the EC2 instance is terminated the instance sends a message to the controlling instance to ensure that the terminating instance is removed from the cluster
3. If the EC2 instance is started at a particular IP address it creates the cluster and acts as the controlling instance of the cluster

This is a very simple setup but will serve many applications. For more complex needs you'll need to write your own tools. Contact support@franz.com to discuss support options.

The choice of AMI on which to build our AMI is not important except that our scripts assume that the initial account name of the image is ec2-user. Thus we suggest that you use one of the Amazon Linux images. If you use another kind of image you'll need to do extra work (as an example we describe below how to use a Centos AMI). Since the instance we'll build with the AMI are used only for AllegroGraph and not for other uses there's no point in running a different version of Linux that you may use in your development work.

These are the steps to build an AMI:

Start an instance using an Amazon Linux AMI with EBS support.

We can't specify the exact name of the image to start as the names change over time and depending on the region. We will usually pick one of the first images listed.

You don't need to start a large virtual machine. A t2.micro will do.

You'll need to specify a VPC and subnet. There should be a default VPC available. If not you'll have to create one.

Make sure that when you specify that subnet that you want to external IP address.

Copy an agraph distribution (tar.gz format) to the ec2 instance into the home directory of ec2-user. Also copy the file aws-repl/aws-repl.tar to the home directory of ec2-user on the instance. aws-repl.tar contains scripts to support replication setup on AWS.

Extract the agraph repo in a temporary spot and run install-agraph in it, specifying the root of the agraph distribution.

I put it in /home/ec2-user/agraph

For example:

```
% mkdir tmp
% cd tmp
% tar xfz ../agraph-6.5.0-linuxamd64.64.tar.gz
% cd agraph-6.5.0
% ./install-agraph ~/agraph
```

Edit the file ~/agraph/lib/agraph.cfg and add the line

UseMainPortForSessions yes

This will allow sessions to be tracked through the Load Balancer.

If you have an agraph license key you should add it to the agraph.cfg file.

Unpack and install the aws-repl code:

```
% tar xf aws-repl.tar
% cd aws-repl
% sudo ./install.sh
```

You can delete aws-repl.tar but don’t delete the aws-repl directory. It will be used on startup.

Look at aws-repl/var.sh to see the parameter values. You’ll see an agraphroot parameter which should match where you installed agraph.

At this point the instance is setup.

You should go to the aws console, select this instance, and from the Action menu select “Image / Create Image”. Wait for the AMI to be built. At this time you can terminate the ec2 instance.

Using a CentOS 7 image:

If you wish to install on top of CentOS then you’ll need additional steps. The initial user on CentOS is called ‘centos’ rather than ‘ec2-user’. In order to keep things consistent we’ll create the ec2-user account and use that for running agraph just as we do for the Amazon AMI.

ssh to the ec2 vm as centos and do the following to create the

ec2-user account and to allow ssh access to it just like the centos account

```
[centos@ip-10-0-1-227 ~]$ sudo sh
```

```
sh-4.2# adduser ec2-user
sh-4.2# cp -rp .ssh ~ec2-user
sh-4.2# chown -R ec2-user ~ec2-user/.ssh
sh-4.2# exit
```

```
[centos@ip-10-0-1-227 ~]
```

```
$
```

At this point you can copy the agraph distribution to the ec2 vm. Scp to ec2-user@x.x.x.x rather than centos@x.x.x.x. Also copy the aws-repl.tar file.

The only change to the procedure is when you must run install.sh in the aws-repl directory.

The ec2-user account does not have the ability to sudo. So this command must be run

when logged in as the user centos;

```
centos@ip-10-0-1-227 ~]$ sudo sh
sh-4.2# cd ~ec2-user/aws-repl
sh-4.2# ./install.sh
```

```
+ cp joincluster /etc/rc.d/init.d
+ chkconfig --add joincluster
sh-4.2# exit


[centos@ip-10-0-1-227 ~]


$
```

Edit the terraform file we supply to suit your needs

Edit the file agelb.tf. This file contains directives to terraform to create the cluster with load balancer. At the top are the variables you can easily change. Other values are found inside the directives and you can change those as well.

Two variables you definitely need to change are

1. “**ag-elb-ami**” – this is the name of the AMI you created in the previous step or the AMI we supply.
2. “**ssh-key**” – this is the name of the ssh key pair you want to use in the instances created.

You may wish to change the region where you want the instances built (that value is in the provider clause at the top of the file) and if you do you’ll need to change the variable “azs”.

We suggest you try building the cluster with the minimum

changes to verify it works and then customize it to your liking.

Run terraform to build the cluster

To build the cluster make sure your have an ~/.aws/config file with a default entry, such as

```
[default]
aws_access_key_id = AKIAIXXXXXXXXXXXXXX
aws_secret_access_key = o/dyrxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

This is what terraform uses as credentials when it contacts AWS.

In order to use terraform the first time (or any time you change the provider clause in agelb.tf) run this command

```
% terraform init
```

Terraform will download the files appropriate for the provider you specified.

After that you can build your cluster with

```
% terraform apply
```

And watch the messages. If there are no errors terraform will wait for confirmation from you to proceed. Type yes to proceed, anything else to abort.

After terraform is finished you'll see the address of the load balancer printed.

You can make changes the agelb.tf file and again 'terraform apply ' and terraform will tell you what it needs to do to change things from how they are now to what the agelb.tf file specifies.

To delete everything terraform added type the command

% terraform destroy

And type yes when prompted.

Turn Customer Service Calls into Enterprise Knowledge Graphs

Franz's CEO, Jans Aasman's recent Destination CRM article:

The need for text analytics and speech recognition has broadened over the years, becoming more prevalent and

essential in the sales, marketing, and customer service departments of various types of businesses and industries. The goal is simple for these contact center use cases: provide real-time assistance to human agents interacting with potential customers to close sales, initiate them, and increase customer satisfaction.

Until fairly recently, the rich array of unstructured data encompassing client texts, chats, and phone calls was obscured from contact centers and organizations due to the sheer arduousness of speech recognition and text analytics. When readily integrated into knowledge graphs, however, these same sources become some of the most credible for improving agent interactions and achieving business objectives.

Powered by the shrewd usage of organizational taxonomies, machine learning, natural language processing (NLP), and semantic search, knowledge graphs make speech recognition and text analytics immediately accessible, enabling real-time customer interactions that can maximize business objectives—and revenues.

Taxonomies

Taxonomies are the foundation of the knowledge graph approach to rapidly conveying results of speech recognition and text analytics for timely customer interactions. Agents need three types of information to optimize customer interactions: their personas (such as an executive or a purchase department representative, for example), their reasons for contacting them, and their industries. Taxonomies are instrumental to performing these functions because they provide a hierarchy of relevant terms to organizations.

Read the full article at Destination CRM

AllegroGraph Named to DBTA Top 100 That Matter Most in Data

Franz Inc., an early innovator in Artificial Intelligence (AI) and leading supplier of Graph and Document Database technology for Knowledge Graphs, today announced that it has been named to Database Trends and Applications (DBTA) – 2019 Top 100 That Matter Most in Data.

"We're excited to announce our seventh annual list, as the industry continues to grow and evolve," remarked Thomas Hogan, Group Publisher at Database Trends and Applications. "Today, more than ever, businesses are looking to increase their efficiency, agility and ability to innovate by managing and leveraging data in new and novel ways. This list seeks to highlight those companies that have been successful in establishing themselves as unique resources for data professionals and stakeholders."

"We are honored to receive this acknowledgement for our efforts in delivering Enterprise Knowledge Graph Solutions," said Dr. Jans Aasman, CEO, Franz Inc. "In the past year, we have seen demand for Enterprise Knowledge Graphs take off across industries along with recognition from top technology analyst firms that Knowledge Graphs provide the critical foundation for artificial intelligence applications and predictive analytics. Our AllegroGraph Knowledge Graph Platform Solution offers a unique comprehensive approach for helping companies accelerate the creation of Enterprise Knowledge Graphs that deliver new value to their organization."

Franz's Knowledge Graph Platform Solution includes both technology and services for building industrial strength Knowledge Graphs based on best-of-class tools, products, knowledge, skills and experience. At the core of the solution is Franz's graph database technology, AllegroGraph, which is utilized by dozens of the top F500 companies worldwide and enables businesses to extract sophisticated decision insights and predictive analytics from highly complex, distributed data that cannot be uncovered with conventional databases.

Franz delivers the expertise for designing ontology and taxonomy-based solutions by utilizing standards-based development processes and tools. Franz also offers data integration services from siloed data using W3C industry standard semantics, which can then be continually integrated with information that comes from other data sources. In addition, the Franz data science team provides expertise in custom algorithms to maximize data analytics and uncover hidden knowledge.

Companies Across the Globe Use Franz Knowledge Graph Solutions

Organizations in customer service, healthcare, life science, publishing and technology have relied on Franz to help develop their knowledge graph solutions.

Global B2B technology firm N3 Results has utilized Franz's Knowledge Graph Solution to build an 'Intelligent Sales Organization,' which 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.

"In a typical sales organization, the valuable content within the online chat or voice conversation between the agent and customer goes into a black hole," said Shannon Copeland, COO of N3. "Franz helped us build a modern Intelligent Sales Organization (ISO) by creating a real-time Knowledge Graph

that knows everything about customers and agents and provides the raw data for machine learning to improve doing the business of ISO. Now we use the rich information between agents and customers to improve the quality of the interaction in real time, which ultimately creates more sales and provides far better analytics for management."

In 2015, Dr. Parsa Mirhaji, his colleagues and industry partners, including Franz Inc. embarked on a project to bring Knowledge Graph technology to Montefiore, a Bronx-based medical center. "Our strategy at Montefiore is to build a data-driven and evidence-based health system – essentially a learning healthcare system – that can understand its own population thoroughly, understand and improve its practices, and develop the highest quality of services for the people it serves," said Parsa Mirhaji, MD, PhD, Director of the Center for Health Data Innovations at Montefiore and the Albert Einstein College of Medicine. "In order to accomplish that goal, we have created a system that harvests every piece of data that we can possibly find, from our own EMRs and devices to patient-generated data to socioeconomic data from the community. It's extremely important to use anything we can find that can help us categorize our patients more accurately." (Health IT Analytics, At Montefiore, Artificial Intelligence Becomes Key to Patient Care, September 10, 2018)

Wolters Kluwer is using graph analytic techniques to accelerate the knowledge discovery process for its clients. "What we're really interested in is achieving insights that today take a person to analyze and that are prohibitive computationally," said Greg Tatham, Wolters Kluwer CTO of Global Platforms. "We're providing this live feedback. As you're typing, we're providing question and suggestions for you live. AllegroGraph gives us a performant way to be able to just work our way through the whole knowledge model and come up with suggestions to the user in real time." (Datanami, How AI Boosts Human Expertise at Wolters Kluwer, June 6, 2018)

Gartner Identifies Knowledge Graphs and Semantics as Key Technologies for AI

Gartner recently recognized knowledge graphs as a key new technology in both their Hype Cycle for Artificial Intelligence and Hype Cycle for Emerging Technologies. Gartner's Hype Cycle for Artificial Intelligence 2018 states, "The rising role of content and context for delivering insights with AI technologies, as well as recent knowledge graph offerings for AI applications have pulled knowledge graphs to the surface."

Semantics has also been identified by Gartner as critical for effectively utilizing enterprise data assets. "Unprecedented levels of data scale and distribution are making it almost impossible for organizations to effectively exploit their data assets. Data and analytics leaders must adopt a semantic approach to their enterprise data assets or face losing the battle for competitive advantage." (Gartner, How to Use Semantics to Drive the Business Value of Your Data, Guido De Simoni, November 27, 2018) For more information about the Gartner report, visit the Gartner Report Order Page.

About Franz Inc.

Franz Inc. is an early innovator in Artificial Intelligence (AI) and leading supplier of Semantic Graph Database technology with expert knowledge in developing and deploying Knowledge Graph solutions. The foundation for Knowledge Graphs and AI lies in the facets of semantic technology provided by AllegroGraph and Allegro CL. The ability to rapidly integrate new knowledge is the crux of the Knowledge Graph and Franz Inc. provides the key technologies and services to address your complex challenges. Franz Inc. is your Knowledge Graph technology partner.

About Database Trends and Applications

Database Trends and Applications (DBTA), published by Information Today, Inc., is a bimonthly magazine that delivers advanced trends analysis and case studies in data management

and analysis developed by a team with more than 25 years of industry experience. Visit www.dbta.com for subscription information. DBTA also delivers groundbreaking market research exclusively through its Unisphere Research group.

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 webcast to learn more about creating your Intelligent Customer Operations approach

Webinar Recording – Youtube.com/allegrograph

Slides are available in PDF or on Slideshare

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

def opendb(name):
                                                    return ag_connect(name,host="localhost",port=10035,user="test",password="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#dateTime>)
```

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.jsonld").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)
```

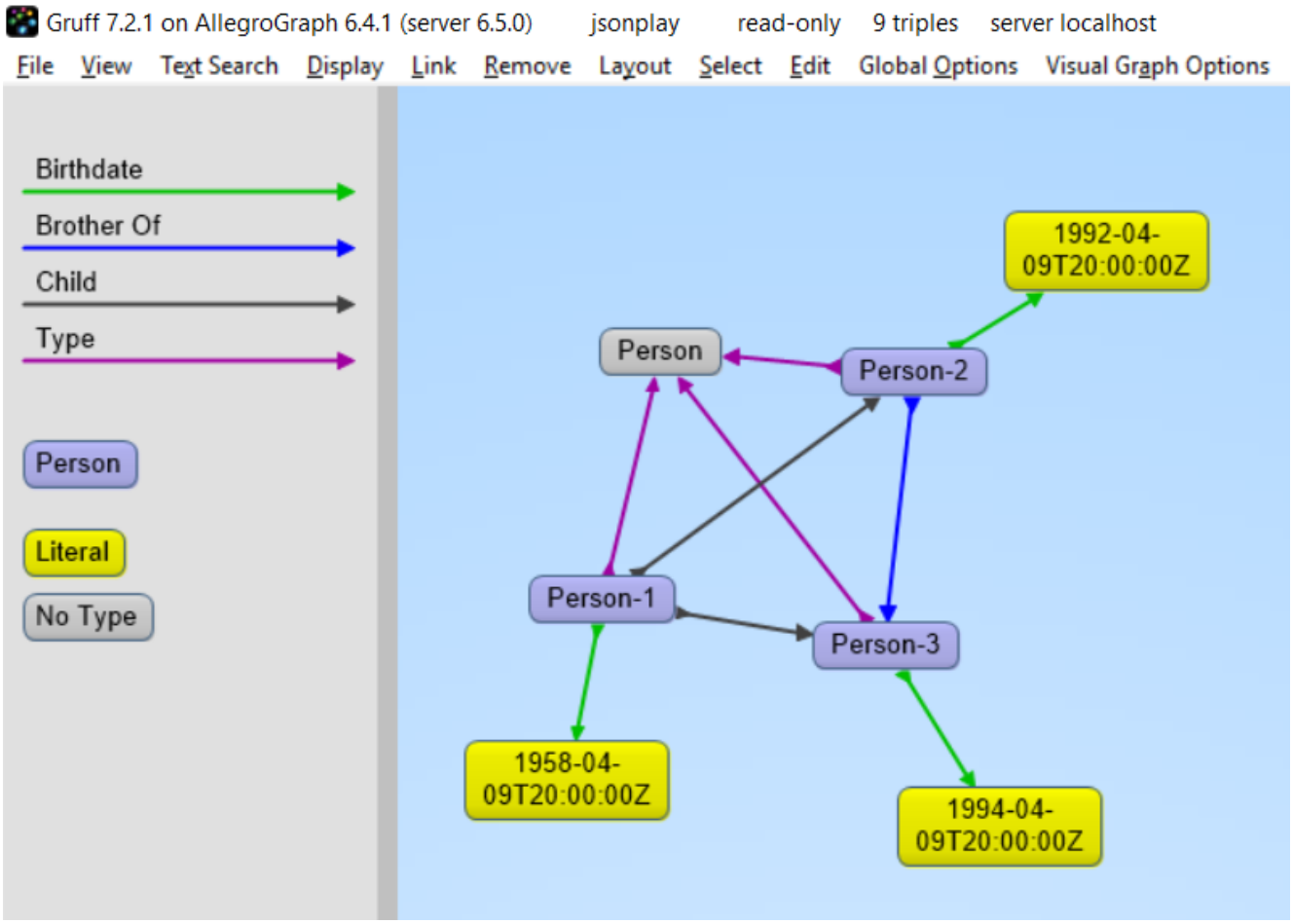
Building a Graph of Objects

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.



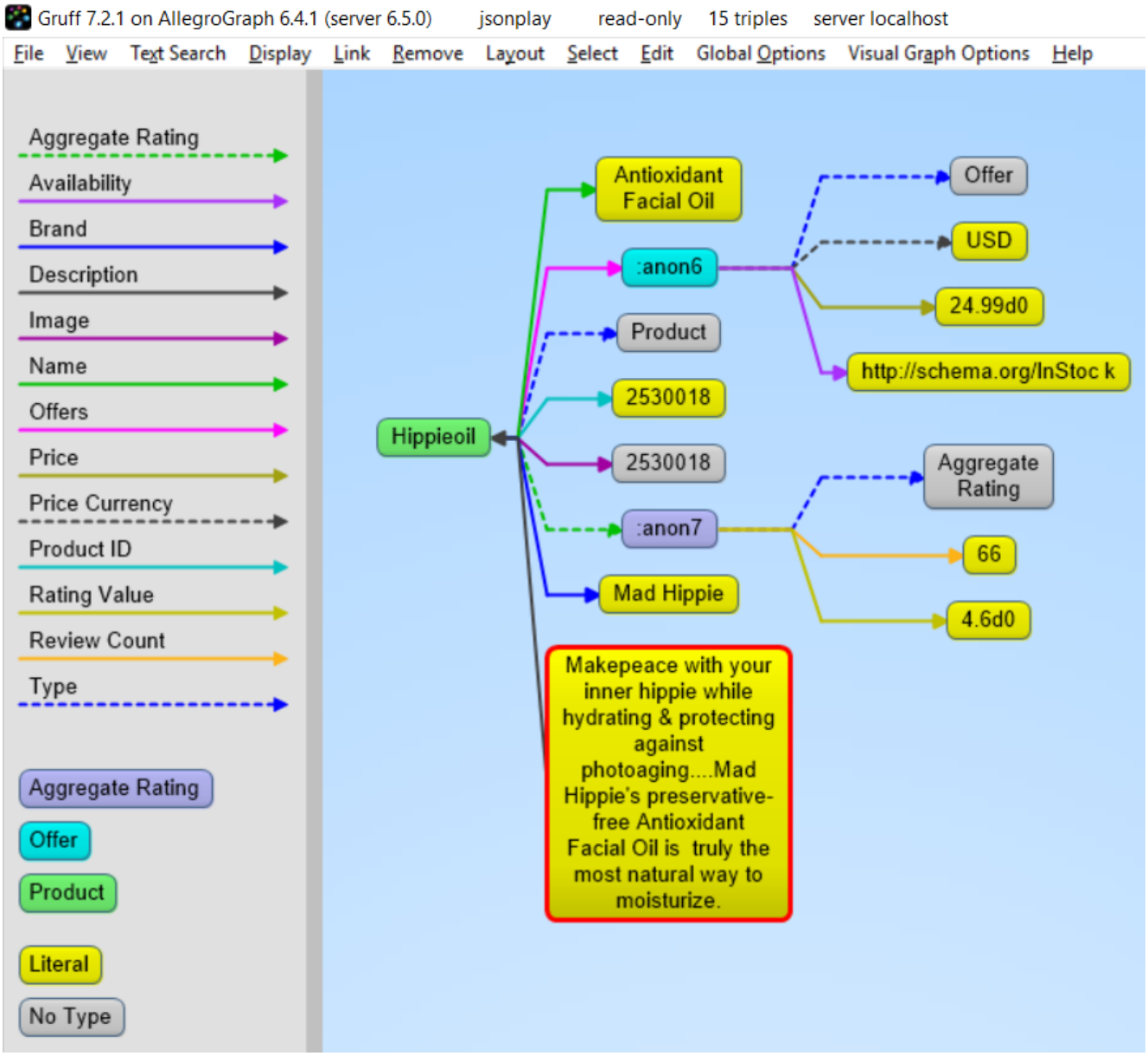
JSON-LD Keyword Directives can be Added at any Level

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

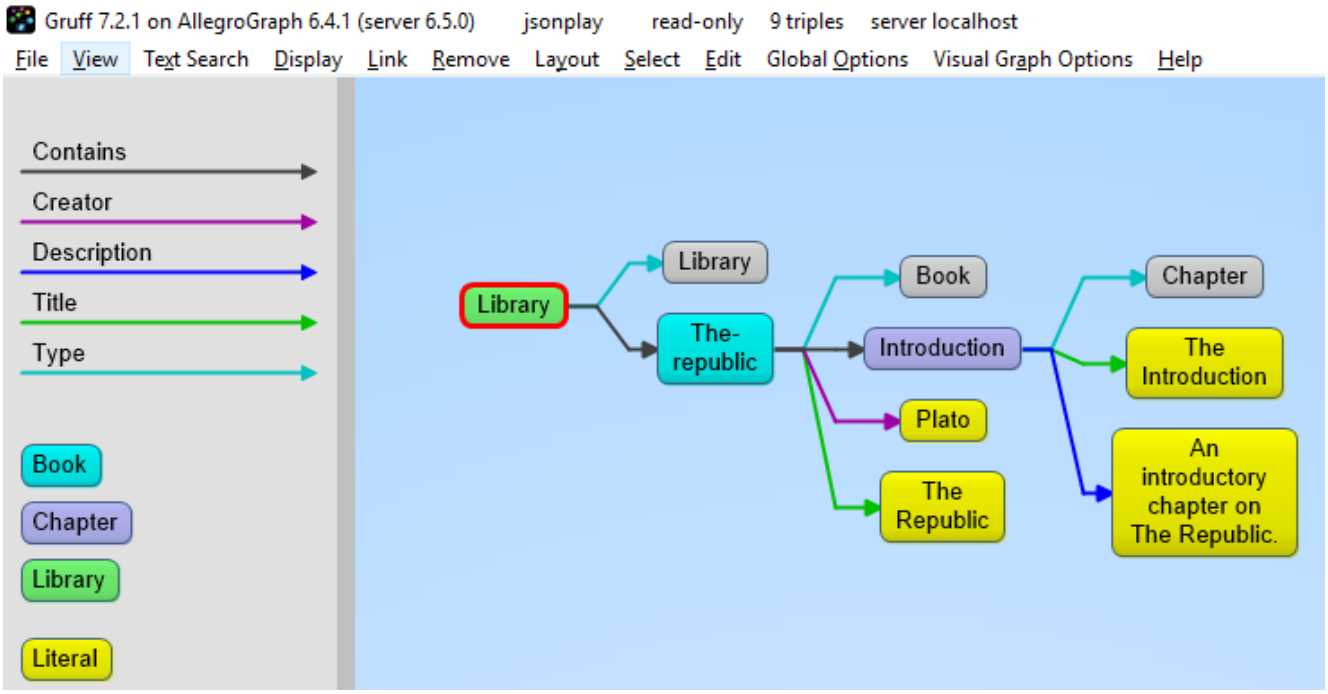
<http://franz.com/place1>. If you don’t add the @id the triples will be put in the default graph.

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.

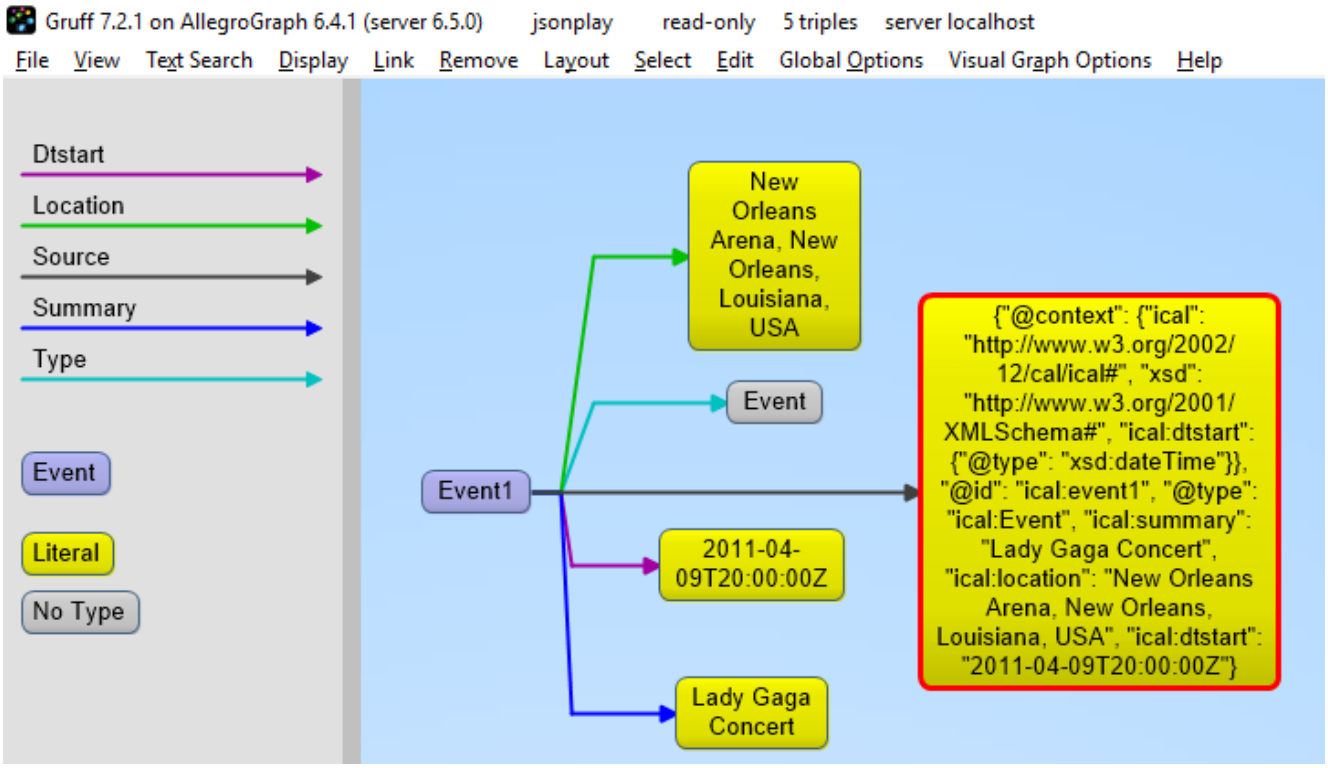
```
event = {
  "@context": {
    "@id": "ical:event1",
    "@type": "ical:Event",
    "ical": "http://www.w3.org/2002/12/cal/ical#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ical:dtstart": { "@type": "xsd:dateTime" }
      },
    "ical:summary": "Lady Gaga Concert",
    "ical:location":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

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