AllegroGraph Tutorial – Distributed Repository Using Shards and Federation Setup

Introduction

A database in AllegroGraph is usually initially implemented as a single repository, running in a single AllegroGraph server. This is simple to set up and operate, but problems arise when the size of data in the repository nears or exceeds the resources of the server on which it resides. Problems can also arise when the size of data fits well within the specs of the database server, but the query patterns across that data stress the system.

When the demands of data or query patterns outpace the ability of a server to keep up, there are two ways to attempt to grow your system: vertical or horizontal scaling.

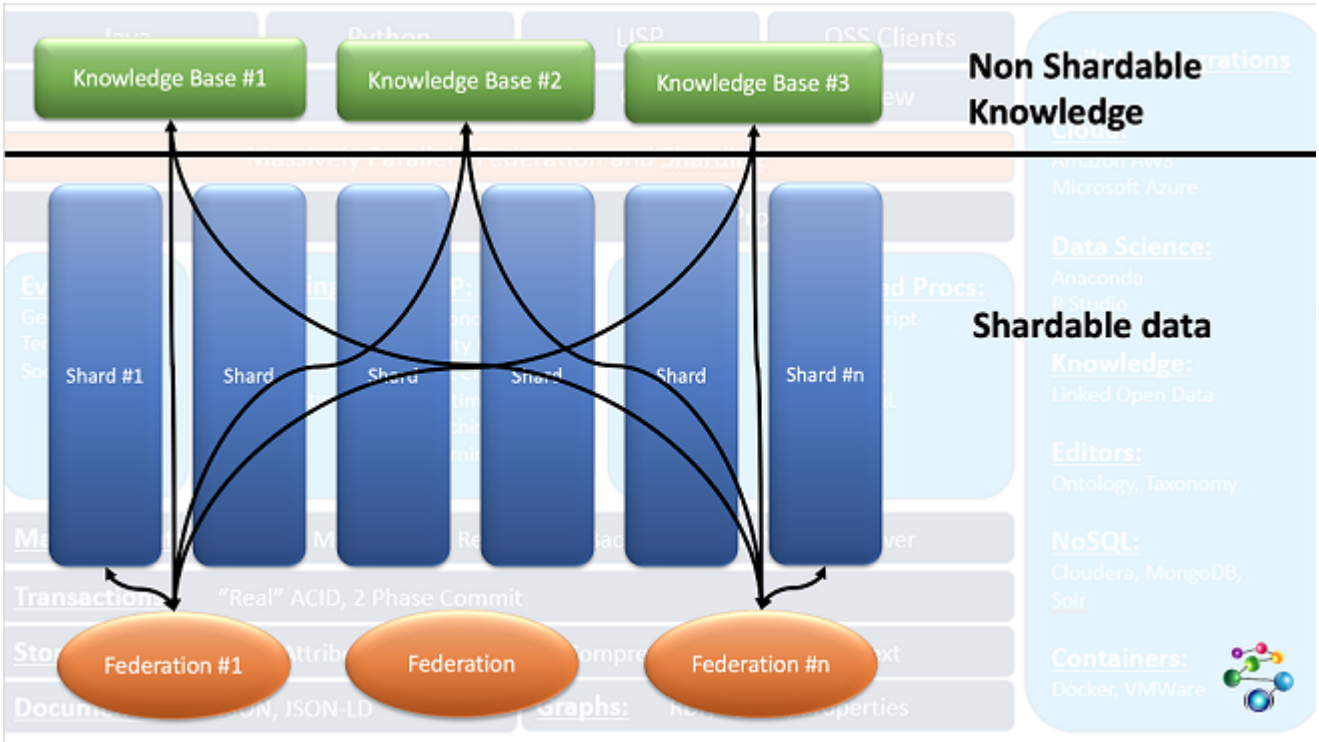
With vertical scaling, you simply increase the capacity of the server on which AllegroGraph is running. Increasing CPU power or the amount of RAM in a server can work for modest size data sets, but you may soon run into the limitations of the available hardware, or the cost to purchase high end hardware may become prohibitive.

AllegroGraph provides an alternative solution: a cluster of database servers, and horizontal scaling through *sharding* and *federation*, which combine in AllegroGraph's FedShard™ facility. An AllegroGraph cluster is a set of AllegroGraph installations across a defined set of machines. A distributed repository is a logical database comprised of one or more repositories spread across one or more of the AllegroGraph nodes in that cluster. A distributed

repository has a required partition key that is used when importing statements. When adding statements to your repository, the partition key is used to determine on which shard each statement will be placed. By carefully choosing your partition key, it is possible to distribute your data across the shards in ways that supports the query patterns of your application.

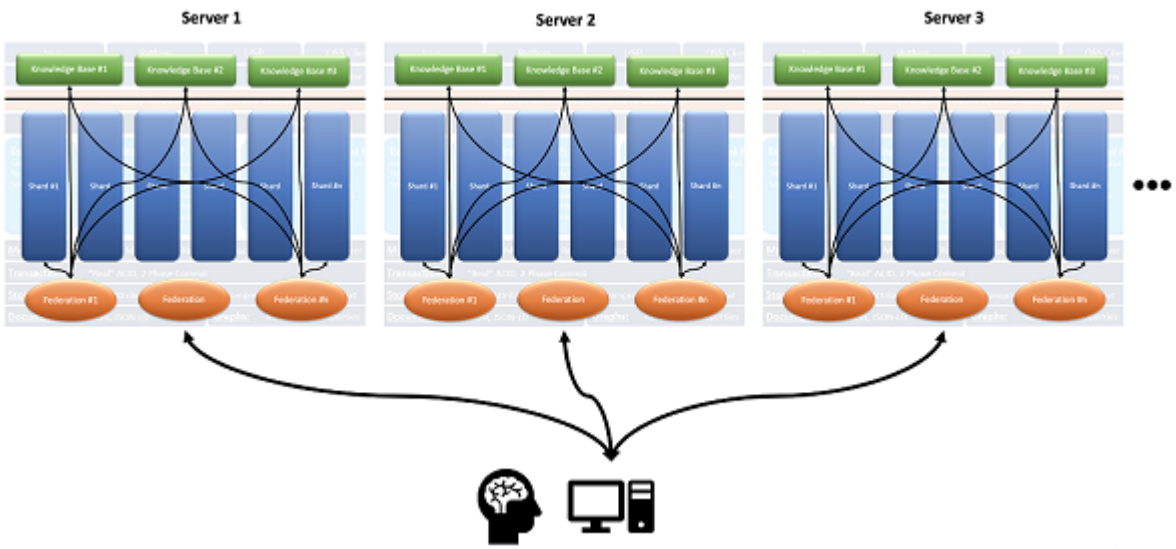
Data common to all shards is placed in knowledge base repositories which are federated with shards when queries are processed. This combination of shards and federated knowledge base repos, called FedShard™, accelerates results for highly complex queries.

This diagram shows how this works:



The three *Knowledge Base* repos at the top contain data needed for all queries. The *Shards* below contain partitionable data. Queries are run on federations of the knowledge base repos with a shard (and can be run of each possible federation of a shard and the knowledge bases with results being combined when the query completes). The black lines show the federations running queries.

The shards need not reside in the same AllegroGraph instance, and indeed need not reside on the same server, as this expanded images shows:



The Distributed Repositories Using Shards and Federation Tutorial walks you through how to define and install to a cluster, how to define a distributed repository, and how various utilities can be used to manipulate AllegroGraph clusters and distributed repositories.

This document describes all the options when setting up a distributed repository (the tutorial just uses some options). The last section, More information on running the cluster, has links into the Tutorial document where things like running a SPARQL query on a cluster are discussed.

The basic setup

You have a very large database and you want to run queries on the database. With the data in a single repository in a single server, queries may take a long time because a query runs on a single processor. At the moment, parallel processing of queries using multiple cores is not supported for a single repository.

But if you can partition your data into several logical groups and your queries can be applied to each group without needing information from any other group, then you can create a distributed repository which allows multiple servers to run queries on pieces of the data effectively in parallel.

Let us start with an example. We describe a simplified version of the database used in the tutorial.

The data is from a hospital. There is diagnosis description data (a list of diseases and conditions) and administration data (a list of things that can happen to a patient while in the hospital – check in, check out, room assignment, etc.) and there is patient data. Over the years the hospital has served millions of patients.

Each patient has a unique identifier, say pNNNNNNNNN, that is the letter p followed by nine decimal digits. Everything that happened to that patient is recorded in one or more triples, such as:

```
p001034027 checkIn          2016-11-07T10:22:00Z
p001034027 seenBy           doctor12872
p001034027 admitted         2016-11-07T12:45:00Z
p001034027 diagnosedHaving condition5678
p001034027 hadOperation     procedure01754
p001034027 checkOut         2016-11-07T16:15:00Z
```

This is quite simplified. The tutorial example is richer. Here we just want to give the general idea. Note there are three objects which refer to other data: condition5678 (broken arm), doctor12872 (Dr. Jones), and procedure01754 (setting a broken bone). We will talk about these below.

So we have six triples for this hospital visit. We also have personal data:

```
p001034027 name             "John Smith"
p001034027 insurance        "Blue Cross"
p001034027 address          "123 First St. Springfield"
```

And then there are other visits and interactions. All in all, there are, say, 127 triples with p001034027 as the subject. And there are 3 million patients, with an average of 70 triples per patient, or 210 million triples of patient data.

Suppose you have queries like:

- How many patients were admitted in 2016?
- How many patients had a broken arm (condition5678)?
- How many broken arm patients were re-admitted within 90 days?
- How many patients stayed in the hospital longer than 2 days?

All of those queries apply to patients individually: that is those questions can be answered for any patient, such as p001034027, without needing to know about any other patient. Contrast that with the query

- What was the next operation in the operating room where p001034027 was treated?

For that query, you need to know when p001034027 used the operating room and what was the next use, which would have been by some other patient. (In the simple scheme described, it is not clear we know which operating room was used and when, but assume that data is in triples not described, all with p001034027 as the subject.) This query is not, in its present form, suitable for a distributed repository since to answer it, information has to be collected from the shard containing p001034027 and then used in retirieving data from other shards.

So if your queries are all of the first type, then your data is suitable for a distributed repository.

Some data is common to all patients: the definition of conditions, doctors, and procedures. You may need to know these data when answering queries. Not if the query is How

many patients were diagnosed with condition5678?' but if it is How many patients had a broken arm? as the latter requires knowing that condition5678` is a broken arm. Thus, triples like

condition5678 hasLabel "broken arm"

are needed on all shards so that queries like

```
SELECT ?s WHERE { ?c hasLabel "broken arm" .
                  ?s diagnosedHaving ?c . }
```

will return results. As we describe, we have an additional repository. the kb (knowledge base) repo which is federated with all shards and provides triples specifying the general taxonomy and ontology.

Resource requirements

The Memory Usage document discusses requirements for repos. Each shard in a distributed repository is a repo so each must have the resources discussed in that document.

Also distributed repositories use many file descriptors, not only for file access but also for pipes and sockets. When AllegroGraph starts up, if the system determines that there may be too few file descriptors allowed, a warning is printed:

```
AllegroGraph Server Edition 7.0.0
Copyright (c) 2005-2020 Franz Inc.  All Rights Reserved.
AllegroGraph contains patented and patent-pending technologies.

Daemonizing...

Server started with warning: When configured to serve a distributed
database we suggest the soft file descriptor limit be 65536. The
current limit is 1024.
```

Cluster Definition File

To support operation over a cluster of servers, AllegroGraph requires a Cluster Definition file named, in the default, *agcluster.cfg*. This file can define distributed repository specifications. We discuss the file in detail below in the agcluster.cfg file section.

The distributed repository setup

A distributed repository has the following components:

- **A set of one of more AllegroGraph servers**. Each server is specified by a host, a scheme (i.e. http or https), and a port. Those three elements uniquely define the server. After installation and cluster setup are complete, AllegroGraph will be installed on each server and will have the cluster repository and one or more cluster shards (a special type of repository) defined in each server. We refer to the servers as *cluster servers*.
- **A distributed repository**. This is a special type of repository. Its name is specified in the *agcluster.cfg* file with the db directive (described below). It appears as a repository on each cluster server but does not itself contain triples. Instead it contains information about the cluster (the servers, the shards, and so on) which is used by the server to manage queries, insertions, and deletions. Queries applied to the distributed repository are applied to each shard and the results and collected and returned, perhaps after some editing and further modification. Distributed repositories are created using specifications in the *agcluster.cfg* file. (To be clear about terminology: the *distributed repository definition* is the whole complex specified by the *agcluster.cfg* file: shards, kb repositories, and the distributed repository.)

- **A set of cluster shards**. A shard is a special type of repository. Shards are named (implicitly or explicitly) in the *agcluster.cfg* file. Shards are created when a distributed repository is created using specifications in the *agcluster.cfg* file. Each shard is created fresh at that time: if there is already a repository on a server which shares the name of a shard, that repository must be superseded (deleted and recreated afresh) when the distributed repository is created.
- **A partition key**. The key identifies which triples belong in the same shard. The key can be a **part**, that is a subject, predicate, object, or graph of a triple, or an attribute name (see the Triple Attributes document). If it is a part, all triples with the same part value are placed in the same shard (all triples have a graph even if it is the default graph so if the key is **part graph**, all triples with the default graph go into the same shard, all with graph *XXX* into the same shard, and so on). For **key attribute attribute-name** all triples with the same value for the attribute with *attribute-name* go into the same shard.
- **The common kb repository or repositories**. These are one or more ordinary repositories which will be federated with each shard when processing a SPARQL query. They are specified in the *agcluster.cfg* file and are associated with the cluster but are otherwise normal repos. In general triples can be added and deleted in the usual manner and queries can be executed as usual unrelated to the ditributed repository. (When a query is run on a distributed repository, the common kb repositories are treated as if read only and so calls to delete triples or SPARQL-DELETE clauses will not delete triples in these common kb repos.) You can have as many common repos as you like and need not have any.

Keep these requirements in mind in the formal descriptions of the directives below.

The agcluster.cfg file

The *agcluster.cfg* file can be used for installation (it can install all the servers and create all the repositories and set up all the necessary mechanics for distributed queries) or it can simply be used for distributed queries after the user has set up everything by hand, or somewhere in between.

agcluster.cfg files contain *directives*. Directive names are case-insensitive, so **Server** is the same as **server**. There are four types of directives:

- **Defaulting directives**: these provide defaults for defining directives and collective directives. See the Defaulting directives section for a complete list. Examples are the **Port** and **Scheme** directives, which provide the default port and scheme values for server directives.
- **Defining directives**: there are two: **server** and **repo**. These define servers and repositories that will make up the distributed repository.
- **Collective directives**: **group** and **db** are the two collective directives. **group** directives define and label collections of servers and repos. **db** directives define actual distributed repositories.
- **Object specification directives**: these directives provide information about specific types of objects, for the most part **db**s and **server**s. They specify aspects (such a username and password for servers, shards per server for **db**s). These are described with the object directives they affect.

The format of an *agcluster.cfg* file is:

Toplevel directives
Collective directives

Comment lines and blank lines may be inserted anywhere in the file.

The toplevel directives can be defaulting directives and defining directives. The defaulting directives provide defaults for any defining directives in the whole file (including those in group and db directives) unless overridden by defaulting directives in the collective directives or specific values in the defining directive. Here is a quick example. (Note we indent directives within the **Group** directive. That is for clarity and has no semantic meaning.)

```
Port 10066
Scheme http
server aghost1.franz.com host1
Group my-group
  Scheme https
  server aghost2.franz.com:12012 host2
  server http://aghost3.franz.com host3
```

Three servers are defined:

- host1 http://aghost1.franz.com:10066 (using toplevel scheme and port defaults)
- host2 https://aghost2.franz.com:12012 (using group scheme default and explicit port value)
- host3 http3://aghost3.franz.com:10066 (using toplevel port default and explicit scheme value)

Toplevel directives are all read when the *agcluster.cfg* file is read and apply to defining directives regardless of whether they are before or after the defaulting directives. (All group directives must be after all toplevel directives.)

If there are duplicate defaulting directives at the toplevel, the last is used and the earlier ones are ignored. So if these directives appear at the toplevel:

```
Port 10035
server http://aghost.franz.com aghost
Port 10066
```

the aghost server is http://aghost.franz.com:10066, using the final port directive, not the first one even though the final one appears after the server directive.

The Distributed Repositories Tutorial has a minimal *agcluster.cfg* file which relies on the system providing default names for all the shard repos. Here is the *agcluster.cfg* file from the tutorial:

```
Port 10035
Scheme http

group my-servers
  server aghost1.franz.com host1
  server aghost2.franz.com host2
  server aghost3.franz.com host3

db bigDB
  key              part graph
  user             test
  password         xyzzy
  shardsPerServer  3
  include          my-servers
```

The file defines:

- Three servers: servers are fully determined by a host (e.g. aghost1.franz.com), a port (10035, specified in a default directive line at the top), and a scheme (http or https, in this case http, specified in a default directive line at the top).
- A group of servers, specified in the group line with the label my-servers.
- A distributed repository named bigDB. This is specified on the db line. A cluster repository with the name bigDB will be visible on each server after the distributed repository is defined and the databases are created.
- The key that will be used to determine which shard a triple is added to. key part graph says assign to a

shard based on the graph of the triple. All triples with the same graph value end up in the same shard.

- A username and password. These should be valid for all servers in the my-servers group.
- An include directive saying the servers in the my-servers group should be used by the distributed repository.
- A shardsPerServer directive saying that each specified server will have three shards.

Comment lines

Comment lines in the *agcluster.cfg* file are lines that start with a #. These are ignored as are blank lines. A # following other text does **not** indicate the remainder of the line is a comment. So

```
# This is a comment
Port 10035 # This is NOT a comment and this line is ill-formed
```

Labels

Many constructs (servers, groups, repos, and db's) can be assigned a label. These labels can be referenced later in the file to refer to the constructs. Some database utilities can also use labels.

A label must precede references to it.

```
# This is OK:
server http://aghost1.franz.com host1

group my-servers
  server host1

# This is NOT OK:
group my-servers
  server host1
```

```
server http://aghost1.franz.com host1
```

All labels exist in the same namespace. Duplicate names are illegal, even when used for different objects:

```
 # This will error:
 repo http://aghost.franz.com/repositories/my-repo label1
 db label1
   [...]
```

Some more simple agcluster.cfg examples

If the agcluster.cfg file just below is used for installation, then all three servers will be installed. When the bigDB distributed repository is the created (with, for example, **agtool create-db**), three shard repositories will be created on each server with names determined by the system. Finally, the distributed cluster repository named bigDB will be accessible on each server.

Now we could have specified more things. For example, we could have specified some of the shard repos:

```
Port 10035
Scheme http

group my-servers
  server aghost1.franz.com host1
  server aghost2.franz.com host2
  server aghost3.franz.com host3

db bigDB
  key              part graph
  repo             host1/repositories/my-shard1
  repo             aghost1.franz.com/repositories/my-shard2
  user             test
  password         xyzzy
  shardsPerServer  3
  include          my-servers
```

We have specified two shard repositories, both on host1, one

using the label host1 and one using the actual host name.

If we use this file to install and create the distributed repository, we will still end up with three servers and three repos, named by the system, on each, and additionally the two named repo shards, for a total of eleven shards.

A note on constructed repository names

As we will describe, when a distributed repository is created, shard repos are often created and named by the system. The names are generated from the repository name and have the following form:

<repository-name>.shard<index>

For example, the shards of a 3-shard distributed repository named distdb will be named distdb.shard0, distdb.shard1 and distdb.shard2 respectively.

But if these names conflict with other existing repository names or with other shard names constructed while the distributed repository is being created, the system will try different names. If it cannot find a suitable name, the distributed repository creation will fail with an error.

The names specified in the examples in this document thus may not correspond to what you actually see, but will usually be pretty close.

The directives in the agcluster.cfg file

Defining directives

The two defining directives are **server** and **repo**.

The server directive

A server is completely specified by a scheme (http or https), a port (a positive integer in the range of acceptable port numbers), and a host. The general format is

server [<scheme>://]host[:<port>] [label]

The <scheme> and <port> can be specified, can come from a defaulting directive, or can be the global default, http for the scheme and 10035 for the port. The label is a name which can be used later in the file to refer to this server.

Here are some examples (we assume no defaulting directives are present except those shown in the examples):

server aghost.franz.com aghost

The server is http://aghost.franz.com:10035 and its label is aghost. The scheme (http) and port (10035) come from the global defaults.

scheme https
port 12001
server aghost1.franz.com aghost1

The server is https//aghost1.franz.com:12001 and its label is aghost1. The scheme (https) and port (12001) come from the defaulting directives just above the server directive.

server http://aghost2.franz.com:13012

The server is http//aghost.franz.com:13012 and it has no label. The scheme, port, and host are all fully specified and use no defaults. (If this is a toplevel directive, it is not very useful as the server cannot be referred to later. Labelless servers can be useful as part of collective

directives as they are used when the collective defined is used. In general, however, it is better to specify a label.

Server-specific directives

The following directives can be specified for a server. They can appear after the **server** directive or as defaulting directives in the current context:

user <username>
    The AllegroGraph user that will be be used when making requests to a server.
password <password>
    The password for the user.

osuser <name> : The username to use when ssh'ing into servers (used by **agraph-control** and **install-agraph** in their respective clustered operation modes).

sudo <boolean>
    [Optional] Allow passwordless **sudo** on each host. **sudo** is necessary if, for example, you wish to install AllegroGraph into a directory that requires root privileges to write to.
bindir <directory>
    The directory where of the *bin/* subdirectory of the directory where AllegroGraph is installed on the server (the installation will be in the parent directory).

The repo directive

A repo (or repository) is completely specified by a server host, a catalog, and a repo name. The general format is

repo <server>[/catalog/<catalog-name>]/repository

<server> can be a SERVER-SPEC or a label of an already defined server. Here are some examples:

server aghost.franz.com aghost

```
repo https://aghost2.franz.com:10077/repositories/my-repo my-repo
repo aghost/catalogs/my-catalog/repositories/cat-repo cat-repo
```

A **repo** directive implicitly defines a server. Thus if either of those **repo** directives appeared as part of a **db** directive (defining a distributed repository) the servers aghost.franz.com (with whatever default values the scheme and port had when the server was defined) and https://aghost2.franz.com:10077 will be included among the distributed repository servers even if there is not a specific server directives including them.

Collective directives: GROUP and DB

There are two types of collections that can be specified in an *agcluster.cfg* file (these are *collective directives*):

- GROUP: a collection of server and/or repo objects, along with default directives that affect elements of the group only.
- DB: a collection of servers and repos where each repo is a shard in a distributed repository. Each repo is associated with a server so this collection must include one or more servers, perhaps defined directly or added with an include statement or specified implicitly in a repo directive. Additional **db**-specific directives may be included (like **shardsPerServer**, all are described below) and defaulting directives that apply to the **db** collection only.

These directives create *contexts* and statements following these directives apply to that context only. All statements up to the next collective directive refer to the context of the current connective directive. Statements that precede any collective directive are *toplevel context* statements.

The GROUP directive

A *group* is a collection of servers defined with **server** directives and/or repos defined with **repo** directives. Groups can be referred to by their labels and included with distributed repositories with the **include** directive (see DB directive below).

The format is

```
group <label>
  [default-directives]
  server-directive <label>
  repo-directive <label>
  include <label of another group>
```

Any number of **server** and **repo** directives can be supplied and in any order. **include** directives includes other groups of servers and repos in this group.

The *label* is needed as otherwise there is no way to refer to the group in other directives.

default-directives are defined below. They usually provide defaults for values in the server and repo directives.

repo-directives are formally defined above but in short are

server-spec-or-label[/catalog/catalog-name]/repositories/repo-name <label>

A *server-spec* is described next. A *server-label* is the label given to a server-directive.

server-directives are formally defined above but in short are

[scheme://][host][:port] <label>

where *scheme* is http (the default) or https, *host* is a hostname (default localhost) and *port* is a port number (default 10035). The *label* is optional. It can refer to the server in other directives. An example is

```
group my-servers
  Port 10650
  Scheme http
  server aghost1.franz.com aghost1
  server https://aghost2.franz.com aghost2
  server aghost3.franz.com:10035 aghost3
```

Defaults are specified for *Port* and *Scheme* and are used when necessary. The servers are (completely specified):

```
  http://aghost1.franz.com:10650 aghost1
  https://aghost2.franz.com:10650 aghost2
  http://aghost3.franz.com:10035 aghost3
```

aghost1 uses both supplied defaults, *aghost2* uses the default port but a different scheme. *aghost3* uses the default scheme but a different port.

Here we include some repo directives

```
group my-shards
  Post 10650
  Scheme http
  server aghost1.franz.com aghost1
  server https://aghost2.franz.com aghost2
  server aghost3.franz.com:10035 aghost3
  repo aghost1/repositories/my-rep1
                                                    repo
http://ag-other-host.franz.com/catalog/shard-cat/repositories/
my-other-repo
```

One repo-directive uses a server label and the other specifies a server (with host ag-other-host.franz.com) not otherwise listed.

The DB directive

The **db** directive defines a collection of repositories and servers which collectively form a distributed repository. Triples in the distributed repository are stored in the individual repositories, which are

called *shards*. kb directives define additional repos which contains things like triples defining the database ontology. These repos are federated with shards during SPARQL queries. Queries are run by each server on each shard and the results are combined and returned as the query result. See the Distributed Repositories Tutorial for information on how distributed repositories work. That document contains a fully worked out example. It also contains a *agcluster.cfg* file, which though quite short and straightforward, allows for rich and complex examples. (While the specification allows for many options and complex configurations, most actual use cases do not require long or complex cluster config files.)

The specification for a **db** directive is as follows:

```
db <label>
  [defaulting directives]
  key             <part-or-attribute> <part-type-or-attribute-
name>
  prefix          <string>
  shardsPerServer <positive integer>
  kb              <repo spec or label>
  include         <group label>
  server          <server spec or label>
  repo            <repo spec or label>
```

key, **prefix**, **shardsPerServer**, and **kb** are DB-specific directives. Here are the directives used above:

- **The db label**: this must be specified. It will name the distributed repository and might be used in naming shards not specifically named.
- **defaulting directives**: see the Defaulting directives section below. These directives can provide default values for other directives. Any number of defaulting directives can be specified.
- **key**: this required directive specifies how triples should be assigned to shards. There are two arguments, the **type** and the **value**. The **type** is

either *part* or *attribute*. The possible **values** for part are **subject**, **predicate**, **object**, and **graph**. The **value** for **attribute** is an attribute name. See the Triple Attributes document. For *key attribute name* all triples loaded into the distributed repository must have the the *name* attribute with a value. (All triples have a subject, predicate, object, and graph, the graph being the default graph is no graph is specified when the triple is added.) Only one key can be specified and once the distributed repository is created, it cannot be changed.

- **prefix**: string to be used when generating unique names for shards. It defaults to the **db** label. Only one prefix can be specified.
- **shardsPerServer**: a positive integer specifying the number of shards per server. The default is 1. Servers need not have the same number of shards and may have more shards than this value, but cannot have fewer. Thus all servers must have at least one shard. This directive can be specified once only.

- **kb**: a repo spec or label. This repository will be federated with each shard when processing a query. This repo typically contains ontology data and data which provides information elements of triples. Multiple kb directives can be specified.
- **include**: a group label. The servers in the group will be used in the distributed repository. Additional servers can be designated with the next directive. Multiple **include** directives can be specified.
- **server**: a server spec or label. This server will be included in the distributed repository. Multiple **server** directives can be specified. Each **server** declaration will result in **shardsPerServer** shards being added, with names constructed from the **prefix** or the **db** label, without

regard to **repo** declarations even if the repository spec supplied specifies the same server as a **server** directive. See the example below.

- **repo**: a repository spec or label. The repository will be included in the distributed repository. Multiple **repo** directives can be specified. **repo** directives add shards but they are not counted as the in shardsPerServer value, See the example below.

Here are some examples. Suppose we have this group directive:

```
group my-servers
  port 10035
  scheme http
  server aghost1.franz.com host1
  server aghost2.franz.com host2
  server aghost3.franz.com host3
```

Here is a db directive with the servers specified with and include directive`:

```
db my-cluster
   include my-servers
```

Here is an equivalent db directive specifying servers directly:

```
db my-cluster
  port 10035
  scheme http
  server aghost1.franz.com host1
  server aghost2.franz.com host2
  server aghost3.franz.com host3
```

Here the servers do not all use the same port or scheme. First we create a **group**:

```
group my-servers
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
```

```
db my-cluster
  include my-servers
```

Here is the same **db** with the servers specified directly:

```
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
```

All of those *db* above define a distributed repository with three shards (since *shardsPerServer* defaults to 1) with shard name on each server *my-cluster.shard0*. If we specified a prefix:

```
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
  prefix mc-shard
```

The shard name on each server would be *mc-shard.shard0*.

Here we indicate that each server will have 3 shards with names created using the **db** label (*my-cluster*):

```
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
  shardsPerServer 3
```

This directive will result in 9 shards (3 for each server) named, on each server *my-cluster.shard0*, *my-cluster.shard1*, *my-cluster.shard2*. Here is a **db** directive where some shards are named directly:

```
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  shardsPerServer 3
  repo http://aghost3.franz.com:10044/repositories/my-h1-repo1
  repo http://aghost3.franz.com:10044/repositories/my-h1-repo2
```

This directive will also result in eight shards, 6 (3 in host1 and 3 in host2) named by the system (with names *my-cluster.shard0*, *my-cluster.shard1*, *my-cluster.shard2*) and the two repos on aghost3.franz.com. Because aghost3.franz.com does not appear in a **server** declaration, it only gets the 2 shards specified by the **repo** declarations and no additional shards are created in that server.

Here we specify *shardsPerServer* to be 3 but also specify a fourth repo in *host1*. We end up with 10 shards:

```
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
  shardsPerServer 3
  repo host1/repositories/my-h1-repo4
```

This can be a little confusing but the rule is: for each **server** declaration in the **db** context, **shardsPerServer** shards will be created, named with names constructed from the **prefix** or the **db** label if no **prefix** is specified. Then any **repo** directives will result in additional shards. So

```
db my-cluster
  server http://agraph1.franz.com/
  server http://agraph2.franz.com/
  repo http://agraph1.franz.com/repositories/my-repo1
  repo http://agraph2.franz.com/repositories/my-repo1
  server http://agraph3.franz.com/
```

will results in the following 5 shards (since **shardsPerServer** is unspecified, its value is 1 (the default):

```
http://agraph1.franz.com/repositories/my-cluster.shard0
http://agraph2.franz.com/repositories/my-cluster.shard0
http://agraph1.franz.com/repositories/my-repo1
http://agraph2.franz.com/repositories/my-repo1
http://agraph3.franz.com/repositories/my-cluster.shard0
```

When a server declarations is at the toplevel and not part of the **db** context, it does not get additional shards even though repos in it are made into shards:

```
server http://agraph1.franz.com/ host1
server http://agraph2.franz.com/ host2
db my-cluster
  repo host1/repositories/my-repo1
  repo host2/repositories/my-repo1
  server http://agraph3.franz.com/
```

results in these shards:

will results in the following 3 shards:

```
http://agraph1.franz.com/repositories/my-repo1
http://agraph2.franz.com/repositories/my-repo1
http://agraph3.franz.com/repositories/my-cluster.shard0
```

The **kb** directive: Here we specify a repo as the value of the **kb** directive. This repo will be federated with each shard when processing a query.

```
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
  shardsPerServer 3
                                                    kb
https://my-server.franz.com:10022/catalog/kb-cat/repositories/
my-kb
```

Equivalently, we can specify the server at the toplevel with a label and use the label in the **kb** directive:

```
server  https://my-server.franz.com:10022 my-kb-server
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
  shardsPerServer 3
  kb my-kb-server/catalog/kb-cat/repositories/my-kb
```

We cannot define the my-kb-server under the **db my-cluster** line because then it would be included among the servers with shards. (It is, of courrse, ok to have the **kb** repo on a server with shards, but if we want it on a server without shards, it must be specified on the **kb** line or at the toplevel.)

Equivalently again we can specify the repo at the toplevel with a label and use the label on the **kb** line:

```
server  https://my-server.franz.com:10022 my-kb-server
repo my-kb-server/catalog/kb-cat/repositories/my-kb my-kb-repo
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
  shardsPerServer 3
  kb my-kb-repo
```

of equivalently again:

```
repo
https://my-server.franz.com:10022/catalog/kb-cat/repositories/
my-kb my-kb-repo
db my-cluster
  server http://aghost1.franz.com:10044 host1
  server https://aghost2.franz.com:10035 host2
  server http://aghost3.franz.com:10035 host3
  shardsPerServer 3
  kb my-kb-repo
```

Defaulting directives

These directives provide defaults for resolving *server* and *repo* directives. Values specified in those directives can override the default.

- **port**: the port use by a server. Default when no **port** value is specified is 10035.
- **scheme**: the protocol to use when connecting to a server.

The value must be http or https. Default when no **scheme** is specified in http.

- **catalog**: the catalog to use when resolving repos. Default when no **catalog** is specified is the root catalog.

All directives applicable to servers can also be defaulting directives and can appear at the toplevel (and so affect any group which does not specify a different default and any server or repo which does not specify a different value and which is not in a group with a different default, see examples in the Server directives section).

Here is part of an *agcluster.cfg* file (server and repo specifications are described above):

```
Port 10035
Catalog my-catalog
Scheme https

Server aghost1.franz.com host1
Repo host1/repositories/my-repo repo1
```

The full server specification is

https://aghost1.franz.com:10035

The full repo specification is

https://aghost1.franz.com:10035/catalog/my-catalog/repositories/my-repo

Default values from the scheme, port, and catalog were filled in because of the toplevel defaulting directives.

The following server directives can also appear at the toplevel (these are documented above:

- **user** <username>
- **password** <password>
- **osuser** <name>

- **sudo** <boolean>
- **bindir** <directory>

Installing AllegroGraph on multiple servers

The clustering support in AllegroGraph is designed to allow you to work with all servers with few or even single commands. It is strongly recommended that you arrange things so you use the same directories on each server and use the same scheme, port and username and password. All those must be specified in the *agraph.cfg* file, so then the same *agraph.cfg* file will work on all the servers. The user must have sufficient permissions to perform operations on the servers and distributed repositories (superusers/administrators typically have all necessary permissions). These things can be different on each server but that requires having separate *agraph.cfg* files and a more complex *agcluster.cfg* file and makes simultaneous installation on multiple servers difficult or impossible.

If you have the same *agraph.cfg* file for all servers, the following command will install on all servers defined in the *agcluster.cfg* file and copy the *agcluster.cfg* and *agraph.cfg* files to each AllegroGraph installation:

```
install-agraph --cluster-config agcluster.cfg --agraph-config agraph.cfg [install-dir]
```

install-agraph is located in the untarred AlllegroGraph distribution directory. It and the two *.cfg* files must have paths supplied so the system knows where they are.

If you have a **bindir** directive in the *agcluster.cfg*, the **install-dir** argument can be left out as it can be inferred from the **bindir** value. If **install-dir** is specified, it must be

an absolute pathname.

The distributed AllegroGraph installation process generates a number of temporary files. These are placed in /tmp unless a --staging-dir argument is supplied to the **install-agraph** call. The value can either be an absolute pathname which names a directory which must be accessible on every host in the cluster. All temporary files are removed when the installation completes. If temporary file space runs out, the installation will fail and all new installations will be deleted.

If you cannot use the same directories, schemes, ports, or superuser/passwords on all servers, then install AllegroGraph on each server and run

install-agraph --cluster-config agcluster.cfg [install-dir]

That will cause the **agcluster.cfg** file to be copied around. That file should have the varying specifications for each server.

Changing the agcluster.cfg file

If you want to add information to the *agcluster.cfg* file (to, for example, add distributed repository – **db** –specifications), simply update a copy of *agcluster.cfg* (in one the *lib/* subdirectories of one of the distributed repo servers) and run

install-agraph –cluster-config [path of modified copy/]agcluster.cfg

That will copy the revised file to the various installations on the servers.

Be careful not to modify the specifications for distributed repos already created. The **install-dir** argument is not needed since a **bindir** directive was added to the *agcluster.cfg* file when it was copied to the *lib/* subdirectory of the

installation directories on the various servers.

Starting and stopping the servers

Servers can be started and stopped in the usual way, with commands like

agraph-control --agraph-config <agraph.cfg file> start/stop

All servers in a cluster can be started and stopped with

agraph-control --cluster start/stop

when the invoked **agraph-control** program is in the *bin/* directory of one of the server installations (because it then knows how to find the *agcluster.cfg* file). If **agraph-control** is from somewhere else or does not find the file as expected, specify the location of the file with

agraph-control --cluster-config <agcluster.cfg file> start/stop

Using agtool utilities on a distributed repository

The agtool General Command Utility has numerous command options that work on repositories. Most of these work on distributed repositories just as they work on regular repositories. But here are some notes on specific tools.

Using agtool export on a distributed repository

agtool export (see Repository Export) works on distributed repositories just like it does with regular repositories. All data in the various shards of the distributed repo is written

to a regular data file which can be read into a regular repository or another distributed repo with the same number of shards or a different number of shards. (Nothing in the exported file indicates that the data came from a distributed repository).

The **kb** (knowledge base) repos associated with a distributed repo (see above in this document) are repos which are federated with shards when SPARQL queries are being processed. **kb** repos are not exported along with a distributed repo. You must export them separately if desired.

Using agtool archive on a distributed repository

The **agtool archive** command is used for backing up and restoring databases. For backing up, it works similarly to backing up a regular repo (that is, the command line and arguments are essentially the same).

But a backup of a distributed repo can only be restored into a distributed repo with the same number of shards. It cannot be restored into a regular repo or into a distributed repo with a different number of shards. So for example, suppose we have a distributed repo **bigDB** defined as follows in a *agcluster.cfg* file:

```
Port 19700
Scheme http

group my-servers
  server aghost1.franz.com host1
  server aghost2.franz.com host2

db bigDB
  key part graph
  user     test
  password xyzzy
```

```
  shardsPerServer  3
  include my-servers
```

Here is the **agtool archive backup** command:

```
% bin/agtool archive backup
http://test:xyzzy@aghost1.franz.com:19700/repositories/bigDB
/aghost1/disk1/user1/drkenn1/
agtool built with AllegroGraph Server
Opening triple store bigDB for backup to
/aghost1/disk1/user1/drkenn1/archives/root/bigDB/bigDB.agbacku
p
Backup throughput: 20.9 MB/s
Backup completed in 0h0m3s
Wrote 62.8 MiB to
/aghost1disk1/user1/drkenn1/archives/root/bigDB/bigDB.agbackup
```

The backup went fine. But if we try to restore to a regular repo or to a distributed repo with a different number of shards (there are six shards in our example, 3 on each of 2 servers), it will fail.

But we can restore to a different distributed repo with 6 shards, say one specified like this:

```
db restoreDB1
  key part graph
  user test
  password xyzzy
  server aghost1.franz.com
  shardsPerServer 6
```

as follows:

```
% bin/agtool archive restore --newuuid
http://test:xyzzy@aghost1.franz.com:19700/repositories/restore
DB1 /aghost1/disk1/user1/drkenn1/ bigDB
agtool built with AllegroGraph Server
Restoring archive from /aghost1/disk1/user1/drkenn1/ to new
triple-store restorebigDB1
Restore throughput: 0.7 MB/s
Restore completed in 0h1m24s
```

```
Read 62.8 MiB from
/aghost1/disk1/user1/drkenn1/archives/root/bigDB/bigDB.agbacku
p
```

And of course we can restore to a distributed repo with the same server/shard configuration, like **bugDB3** with a spec similar to **bigDB**'s:

```
group my-servers
  server aghost1.franz.com host1
  server aghost2.franz.com host2

db bigDB
  key part graph
  user     test
  password xyzzy
  shardsPerServer  3
  include my-servers

db bigDB3
  key part graph
  user     test
  password xyzzy
  shardsPerServer  3
  include my-servers

% bin/agtool archive restore --newuuid
http://test:xyzzy@aghost1.franz.com:19700/repositories/bigDB3
/aghost1disk1/user1/drkenn1/ bigDB
agtool built with AllegroGraph
Restoring archive from /aghost1/disk1/user1/drkenn1/ to new
triple-store bigDB3
Restore throughput: 0.8 MB/s
Restore completed in 0h1m20s
Read 62.8 MiB from
/aghost1/disk1/user1/drkenn1/archives/root/bigDB/bigDB.agbacku
p
%
```

Upgrading to a new version

Upgrading to a new version is described in the Repository Upgrading document. It works with distributed repos as with regular repos with the exception that the later version must have a sufficiently similar *agcluster.cfg* file, with the same servers and specifications for existing distributed repos as the older version.

Distributed repos in AGWebView

AGWebView is the browser interface to an AllegroGraph server. For the most part, a distributed repo looks in AGWebView like a regular repo. The number of triples (called by the alternate name *Statements* and appearing at the top of the *Repository* page) is the total for all shards and commands work the same as on regular repos.

You do see the difference in **Reports**. In many reports individual shards are listed by name. (The names are assigned by the system and not under user control). Generally you do not act on individual shards but sometimes information on them is needed for problem solving.

More information on running the cluster

See the Distributed Repositories Tutorial for information on using a cluster once it is set up. See particularly the sections:

- Creating the Shards of a Distributed Repository
- Adding data to a distributed repository
- Querying a distributed repository using SPARQL

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#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.json
ld").json()['@context']
>>> context
```

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


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

person = {
  "@context": context,
```

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

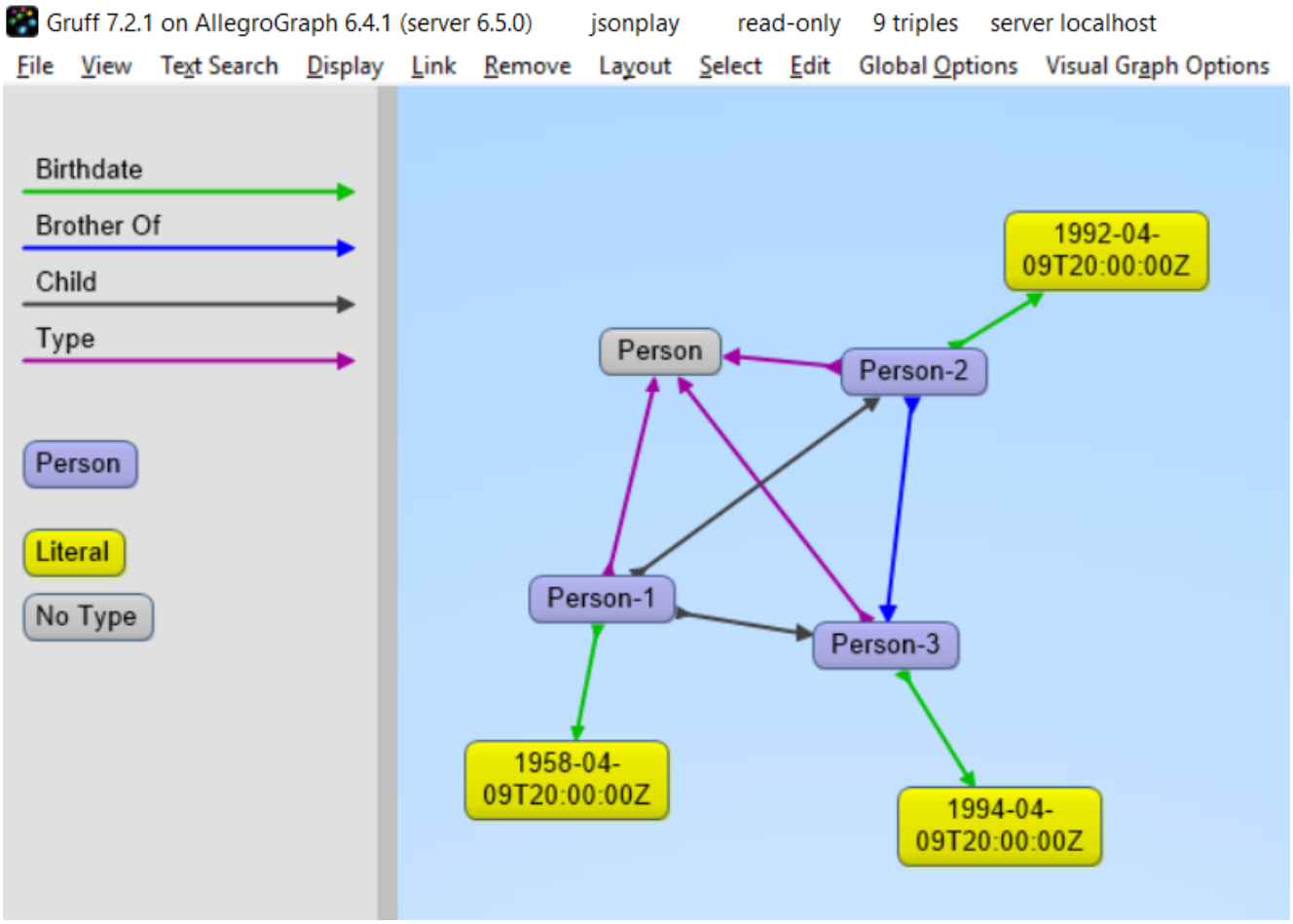
Building a Graph of Objects

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

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

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

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

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

```
}

test([p1,p2,p3])


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

The following shows the graph that we created in Gruff. Note that this is what JSON-LD is all about: connecting objects together.



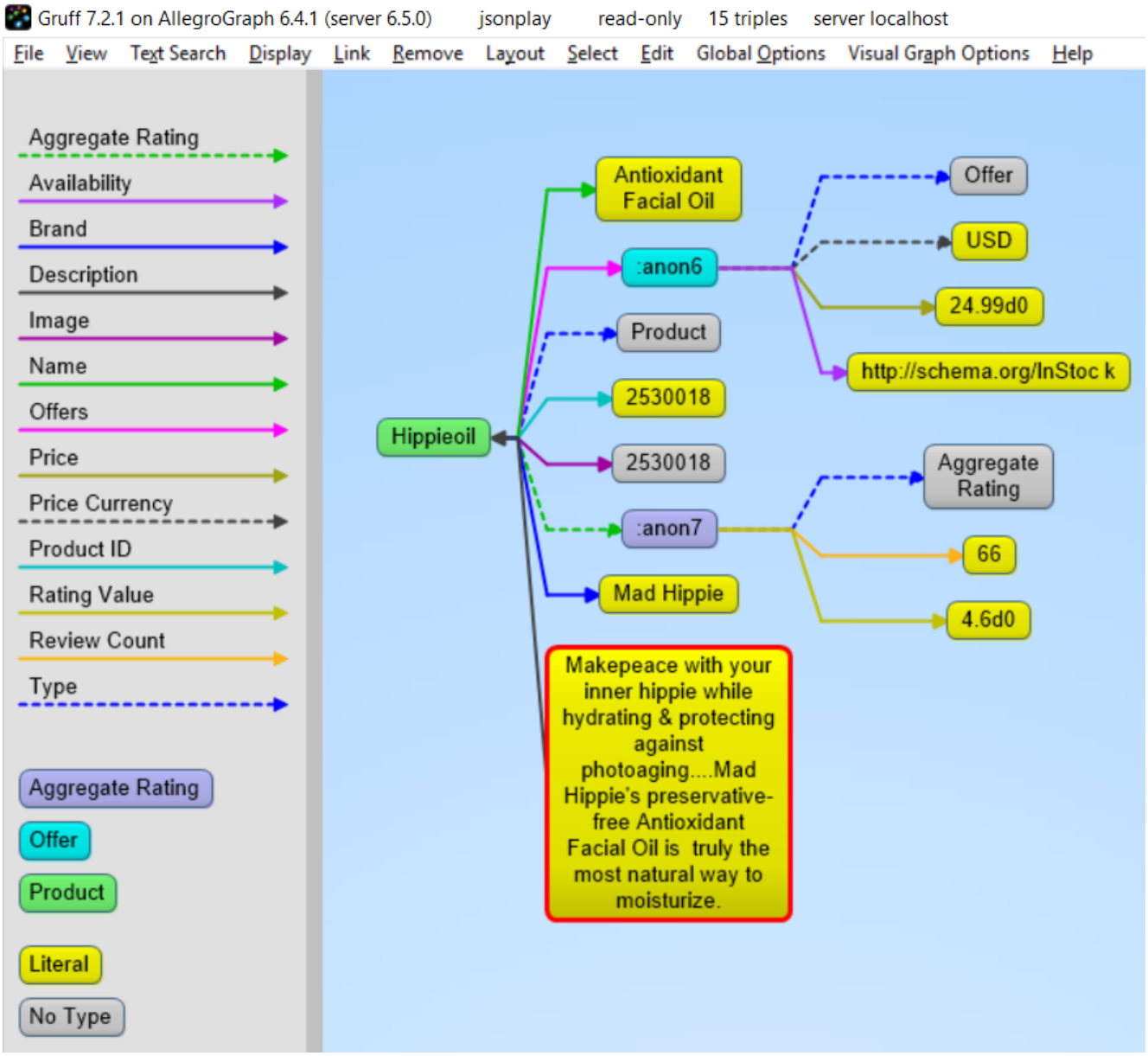
JSON-LD Keyword Directives can be Added at any Level

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

```
hippieoil = {"@context":"http://schema.org",
 "@type":"Product",
 "@id":"http://franz.com/hippieoil",
 "aggregateRating":
    {"@type":"AggregateRating",
     "ratingValue":4.6,
```

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

test(hippieoil)



JSON-LD @graphs

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

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

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

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

and here is the result:

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

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

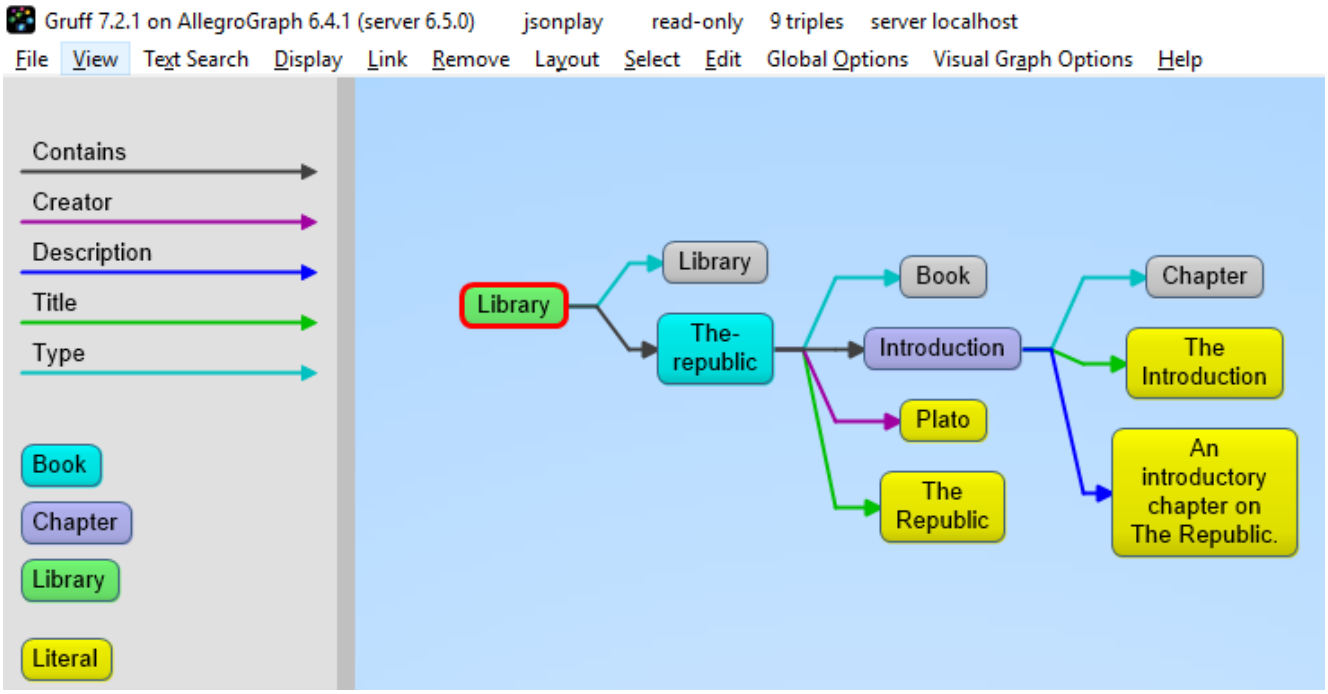
<http://franz.com/place1>. If you don’t add the @id the triples will be put in the default graph.

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.

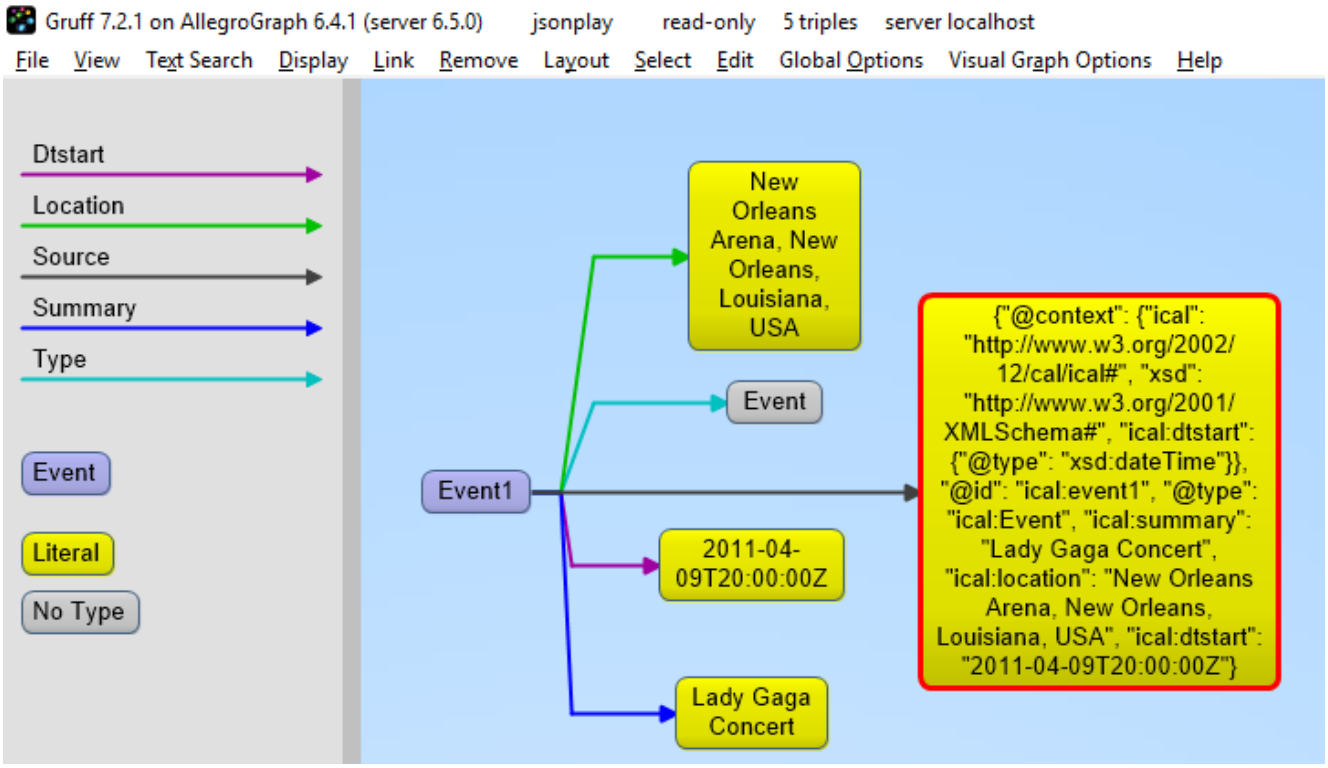
```
event = {
  "@context": {
    "@id": "ical:event1",
    "@type": "ical:Event",
    "ical": "http://www.w3.org/2002/12/cal/ical#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ical:dtstart": { "@type": "xsd:dateTime" }
      },
    "ical:summary": "Lady Gaga Concert",
    "ical:location":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