

Multi-Master Replication Clusters in Kubernetes and Docker Swarm

For more examples visit –
<https://github.com/franzinc/agraph-examples>

Introduction

In this document we primarily discuss running a Multi-Master Replication cluster (MMR) inside Kubernetes. We will also show a Docker Swarm implementation.

This directory and subdirectories contain code you can use to run an MMR cluster. The second half of this document is entitled *Setting up and running MMR under Kubernetes* and that is where you'll see the steps needed to run the MMR cluster in Kubernetes.

MMR replication clusters are different from distributed AllegroGraph clusters in these important ways:

1. Each member of the cluster needs to be able to make a TCP connection to each other member of the cluster. The connection is to a port computed at run time. The range of port numbers to which a connection is made can be constrained by the `agraph.cfg` file but typically this will be a large range to ensure that at least one port in that range is not in used.
2. All members of the cluster hold the complete database (although for brief periods of time they can be out of sync and catching up with one another).

MMR replication clusters don't quite fit the Kubernetes model in these ways

1. When the cluster is running normally each instance knows

the DNS name or IP address of each other instance. In Kubernetes you don't want to depend on the IP address of another cluster's pod as those pods can go away and a replacement started at a different IP address. We'll describe below our solution to this.

2. Services are a way to hide the actual location of a pod however they are designed to handle a set of known ports.. In our case we need to connect from one pod to a known-at-runtime port of another pod and this isn't what services are designed for.
3. A key feature of Kubernetes is the ability to scale up and down the number of processes in order to handle the load appropriately. Processes are usually single purpose and stateless. An MMR process is a full database server with a complete copy of the repository. Scaling up is not a quick and simple operation – the database must be copied from another node. Thus scaling up is a more deliberate process rather than something automatically done when the load on the system changes during the day.

The Design

1. We have a headless service for our controlling instance StatefulSet and that causes there to be a DNS entry for the name *controlling* that points to the current IP address of the node in which the controlling instance runs. Thus we don't need to hardwire the IP address of the controlling instance (as we do in our AWS load balancer implementation).
2. The controlling instance uses two PersistentVolumes to store: 1. The repo we're replicating and 2. The token that other nodes can use to connect to this node. Should the controlling instance AllegroGraph server die (or the pod in which it runs dies) then when the pod is started again it will have access to the data on those two persistent volumes.
3. We call the other instances in the cluster Copy

instances. These are full read-write instances of the repository but we don't back up their data in a persistent volume. This is because we want to scale up and down the number of Copy instances. When we scale down we don't want to save the old data since when we scale down we remove that instance from the cluster thus the repo in the cluster can never join the cluster again. We denote the Copy instances by their IP addresses. The Copy instances can find the address of the controlling instance via DNS. The controlling instance will pass the cluster configuration to the Copy instance and that configuration information will have the IP addresses of the other Copy instances. This is how the Copy instances find each other.

4. We have a load balancer that allows one to access a random Copy instance from an external IP address. This load balancer doesn't support sessions so it's only useful for doing queries and quick inserts that don't need a session.
5. We have a load balancer that allows access to the Controlling instance via HTTP. While this load balancer also doesn't have session support, because there is only one controlling instance it's not a problem if you start an AllegroGraph session because all sessions will live on the single controlling instance.

We've had the most experience with Kubernetes on the Google Cloud Platform. There is no requirement that the load balancer support sessions and the GCP version does not at this time, but that doesn't mean that session support isn't present in the load balancer in other cloud platforms. Also there is a large community of Kubernetes developers and one may find a load balancer with session support available from a third party.

Implementation

We build and deploy in three subdirectories. We'll describe

the contents of the directories first and then give step by step instructions on how to use the contents of the directories.

Directory ag/

In this directory we build a Docker image holding an installed AllegroGraph. The Dockerfile is

```
FROM centos:7
```

```
#  
# AllegroGraph root is /app/agraph  
#
```

```
RUN yum -y install net-tools iputils bind-utils wget hostname
```

```
ARG agversion=agraph-6.6.0  
ARG agdistfile=${agversion}-linuxamd64.64.tar.gz
```

```
# This ADD command will automatically extract the contents  
# of the tar.gz file  
ADD ${agdistfile} .
```

```
# needed for agraph 6.7.0 and can't hurt for others  
# change to 11 if you only have OpenSSL 1.1 installed  
ENV ACL_OPENSSL_VERSION=10
```

```
# so prompts are readable in an emacs window  
ENV PROMPT_COMMAND=
```

```
RUN groupadd agraph && useradd -d /home/agraph -g agraph  
agraph  
RUN mkdir /app
```

```
# declare ARGs as late as possible to allow previous lines to  
be cached  
# regardless of ARG values
```

```
ARG user  
ARG password
```

```
RUN (cd ${agversion} ; ./install-agraph /app/agraph -- --non-interactive \
    --runas-user agraph \
    --super-user $user \
    --super-password $password )

# remove files we don't need
RUN rm -fr /app/agraph/lib/doc /app/agraph/lib/demos

# we will attach persistent storage to this directory
VOLUME ["/app/agraph/data/rootcatalog"]

# patch to reduce cache time so we'll see when the controlling
instance moves.
# ag 6.7.0 has config parameter StaleDNSRetainTime which
allows this to be
# done in the configuration.
COPY dnspatch.cl /app/agraph/lib/patches/dnspatch.cl

RUN chown -R agraph.agraph /app/agraph
```

The Dockerfile installs AllegroGraph in /app/agraph and creates an AllegroGraph super user with the name and password passed in as arguments. It creates a user *agraph* so that the AllegroGraph server will run as the user *agraph* rather than as *root*.

We have to worry about the controlling instance process dying and being restarted in another pod with a different IP address. Thus if we've cached the DNS mapping of *controlling* we need to notice as soon as possible that the mapping has changed. The dnspatch.cl file changes a parameter in the AllegroGraph DNS code to reduce the time we trust our DNS cache to be accurate so that we'll quickly notice if the IP address of *controlling* changes.

We also install a number of networking tools. AllegroGraph doesn't need these but if we want to do debugging inside the container they are useful to have installed.

The image created by this Dockerfile is pushed to the Docker Hub using an account you've specified (see the Makefile in this directory for details).

Directory agrepl/

Next we take the image created above and add the specific code to support replication clusters.

The Dockerfile is

```
ARG DockerAccount=specifyaccount
```

```
FROM ${DockerAccount}/ag:latest
```

```
#  
# AllegroGraph root is /app/agraph
```

```
RUN mkdir /app/agraph/scripts  
COPY . /app/agraph/scripts
```

```
# since we only map one port from the outside into our cluster  
# we need any sessions created to continue to use that one  
port.
```

```
RUN echo "UseMainPortForSessions true" >>  
/app/agraph/lib/agraph.cfg
```

```
# settings/user will be overwritten with a persistent mount so  
copy
```

```
# the data to another location so it can be restored.
```

```
RUN cp -rp /app/agraph/data/settings/user  
/app/agraph/data/user
```

```
ENTRYPOINT ["/app/agraph/scripts/repl.sh"]
```

When building an image using this Dockerfile you must specify

```
--build-arg DockerAccount=MyDockerAccount
```

where MyDockerAccount is a Docker account you're authorized to push images to.

The Dockerfile installs the scripts repl.sh, vars.sh and accounts.sh. These are run when this container starts.

We modify the agraph.cfg with a line that ensures that even if we create a session that we'll continue to access it via port 10035 since the load balancer we'll use to access AllegroGraph only forwards 10035 to AllegroGraph.

Also we know that we'll be installing a persistent volume at /app/agraph/data/user so we make a copy of that directory in another location since the current contents will be invisible when a volume is mounted on top of it. We need the contents as that is where the credentials for the user we created when AllegroGraph was installed.

Initially the file settings/user/username will contain the credentials we specified when we installed AllegroGraph in first Dockerfile. When we create a cluster instance a new token is created and this is used in place of the password for the test account. This token is stored in settings/user/username which is why we need this to be an instance-specific and persistent filesystem for the controlling instance.

When this container starts it runs repl.sh which first runs accounts.sh and vars.sh.

accounts.sh is a file created by the top level Makefile to store the account information for the user account we created when we installed AllegroGraph.

vars.sh is

```
# constants need by scripts
port=10035
reponame=myrepl
```

```
# compute our ip address, the first one printed by hostname
myip=$(hostname -I | sed -e 's/ .*$/')
```

In vars.sh we specify the information about the repository we'll create and our IP address.

The script repl.sh is this:

```
#!/bin/bash
#
## to start ag and then create or join a cluster
##

cd /app/agraph/scripts

set -x
. ./accounts.sh
. ./vars.sh

agtool=/app/agraph/bin/agtool

echo ip is $myip

# move the copy of user with our login to the newly mounted
volume
# if this is the first time we've run agraph on this volume
if [ ! -e /app/agraph/data/rootcatalog/$reponame ] ; then
    cp -rp /app/agraph/data/user/*
/app/agraph/data/settings/user
fi

# due to volume mounts /app/agraph/data could be owned by root
# so we have to take back ownership
chown -R agraph.agraph /app/agraph/data

## start agraph
/app/agraph/bin/agraph-control --config
/app/agraph/lib/agraph.cfg start

term_handler() {
    # this signal is delivered when the pod is
    # about to be killed. We remove ourselves
    # from the cluster.
```



```

    echo got term signal
    /bin/bash ./remove-instance.sh
    exit
}

sleepforever() {
    # This unusual way of sleeping allows
    # a TERM signal sent when the pod is to
    # die to then cause the shell to invoke
    # the term_handler function above.
    date
    while true
    do
        sleep 99999 & wait ${!}
    done
}

if [ -e /app/agraph/data/rootcatalog/$reponame ] ; then
    echo repository $reponame already exists in this
    persistent volume
    sleepforever
fi

controllinghost=controlling

controllingspec=$authuser:$authpassword@$controllinghost:$port
/$reponame

if [ x$Controlling == "xyes" ] ;
then
    # It may take a little time for the dns record for
    'controlling' to be present
    # and we need that record because the agtool program below
    will use it
    until host controlling ; do echo controlling not in DNS
    yet; sleep 5 ; done
    ## create first and controlling cluster instance
    $agtool repl create-cluster $controllingspec controlling
else
    # wait for the controlling ag server to be running

```

```
                until          curl          -s
http://$authuser:$authpassword@$controllinghost:$port/version
; do echo wait for controlling ; sleep 5; done
```

```
    # wait for server in this container to be running
                until          curl          -s
http://$authuser:$authpassword@$myip:$port/version ; do echo
wait for local server ; sleep 5; done
```

```
    # wait for cluster repo on the controlling instance to be
present
```

```
    until $agtool repl status $controllingspec > /dev/null ; do
echo wait for repo ; sleep 5; done
```

```
    myiname=i-$myip
```

```
    echo $myiname > instance-name.txt
```

```
    # construct the remove-instance.sh shell script to remove
this instance
```

```
    # from the cluster when the instance is terminated.
```

```
    echo $agtool repl remove $controllingspec $myiname >
remove-instance.sh
```

```
    chmod 755 remove-instance.sh
```

```
    #
```

```
    # note that
```

```
    # % docker kill container
```

```
    # will send a SIGKILL signal by default we can't trap on
SIGKILL.
```

```
    # so
```

```
    # % docker kill -s TERM container
```

```
    # in order to test this handler
```

```
    trap term_handler SIGTERM SIGHUP SIGUSR1
```

```
    trap -p
```

```
    echo this pid is $$
```

```
    # join the cluster
```

```
    echo joining the cluster
```

```
        $agtool repl grow-cluster $controllingspec
$authuser:$authpassword@$myip:$port/$reponame $myiname
```

```
fi
```

```
sleepforever
```

This script can be run under three different conditions

1. Run when the Controlling instance is starting for the first time
2. Run when the Controlling instance is restarting having run before and died (perhaps the machine on which it was running crashed or the AllegroGraph process had some error)
3. Run when a Copy instance is starting for the first time. Copy instances are not restarted when they die. Instead a new instance is created to take the place of the dead instance. Therefore we don't need to handle the case of a Copy instance restarting.

In cases 1 and 2 the environment variable *Controlling* will have the value "yes".

In case 2 there will be a directory at `/app/agraph/data/rootcatalog/$reponame`.

In all cases we start an AllegroGraph server.

In case 1 we create a new cluster. In case 2 we just sleep and let the AllegroGraph server recover the replication repository and reconnect to the other members of the cluster.

In case 3 we wait for the controlling instance's AllegroGraph to be running. Then we wait for our AllegroGraph server to be running. Then we wait for the replication repository we want to copy to be up and running. At that point we can grow the cluster by copying the cluster repository.

We also create a script which will remove this instance from the cluster should this pod be terminated. When the pod is killed (likely due to us scaling down the number of Copy instances) a termination signal will be sent first to the process allowing it to run this remove script before the pod completely disappears.

Directory kube/

This directory contains the yaml files that create kubernetes resources which then create pods and start the containers that create the AllegroGraph replication cluster.

controlling-service.yaml

We begin by defining the services. It may seem logical to define the applications before defining the service to expose the application but it's the service we create that puts the application's address in DNS and we want the DNS information to be present as soon as possible after the application starts. In the repl.sh script above we include a test to check when the DNS information is present before allowing the application to proceed.

```
apiVersion: v1
kind: Service
metadata:
  name: controlling
spec:
  clusterIP: None
  selector:
    app: controlling
  ports:
    - name: http
      port: 10035
      targetPort: 10035
```

This selector defines a service for any container with a label with a key app and a value controlling. There aren't any such containers yet but there will be. You create this service with

```
% kubectl create -f controlling-service.yaml
```

In fact for all the yaml files shown below you create the object they define by running

```
% kubectl create -f filename.yaml
```

copy-service.yaml

We do a similar service for all the copy applications.

```
apiVersion: v1
kind: Service
metadata:
  name: copy
spec:
  clusterIP: None
  selector:
    app: copy
  ports:
    - name: main
      port: 10035
      targetPort: 10035
```

controlling.yaml

This is the most complex resource description for the cluster. We use a StatefulSet so we have a predictable name for the single pod we create. We define two persistent volumes. A StatefulSet is designed to control more than one pod so rather than a VolumeClaim we have a VolumeClaimTemplate so that each Pod can have its own persistent volume... but as it turns out we have only one pod in this set and we never scale up. There must be exactly one controlling instance.

We setup a liveness check so that if the AllegroGraph server dies Kubernetes will restart the pod and thus the AllegroGraph server. Because we've used a persistent volume for the AllegroGraph repositories when the AllegroGraph server restarts it will find that there is an existing MMR replication repository that was in use when the AllegroGraph server was last running. AllegroGraph will restart that replication repository which will cause that replication instance to reconnect to all the copy instances and become part of the cluster again.

We set the environment variable Controlling to yes and this causes this container to start up as a controlling instance (you'll find the check for the Controlling environment

variable in the repl.sh script above).

We have a volume mount for /dev/shm, the shared memory filesystem, because the default amount of shared memory allocated to a container by Kubernetes is too small to support AllegroGraph.

```
#
# stateful set of controlling instance
#

apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: controlling
spec:
  serviceName: controlling
  replicas: 1
  template:
    metadata:
      labels:
        app: controlling
    spec:
      containers:
        - name: controlling
          image: dockeraccount/agrepl:latest
          imagePullPolicy: Always
          livenessProbe:
            httpGet:
              path: /hostname
              port: 10035
            initialDelaySeconds: 30
          volumeMounts:
            - name: shm
              mountPath: /dev/shm
            - name: data
              mountPath: /app/agraph/data/rootcatalog
            - name: user
              mountPath: /app/agraph/data/settings/user
      env:
        - name: Controlling
```

```

        value: "yes"
volumes:
  - name: shm
    emptyDir:
      medium: Memory
volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      resources:
        requests:
          storage: 20Gi
      accessModes:
        - ReadWriteOnce
  - metadata:
      name: user
    spec:
      resources:
        requests:
          storage: 10Mi
      accessModes:
        - ReadWriteOnce

```

copy.yaml

This StatefulSet is responsible for starting all the other instances. It's much simpler as it doesn't use Persistent Volumes

```

#
# stateful set of copies of the controlling instance
#

apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: copy
spec:
  serviceName: copy
  replicas: 2
  template:
    metadata:

```

```

labels:
  app: copy
spec:
  volumes:
    - name: shm
      emptyDir:
        medium: Memory
  containers:
    - name: controlling
      image: dockeraccount/agrepl:latest
      imagePullPolicy: Always
      livenessProbe:
        httpGet:
          path: /hostname
          port: 10035
          initialDelaySeconds: 30
      volumeMounts:
        - name: shm
          mountPath: /dev/shm

```

controlling-lb.yaml

We define a load balancer so applications on the internet outside of our cluster can communicate with the controlling instance. The IP address of the load balancer isn't specified here. The cloud service provider (i.e. Google Cloud Platform or AWS) will determine an address after a minute or so and will make that value visible if you run

```
% kubectl get svc controlling-loadbalancer
```

The file is

```

apiVersion: v1
kind: Service
metadata:
  name: controlling-loadbalancer
spec:
  type: LoadBalancer
  ports:
    - port: 10035
      targetPort: 10035

```



```
selector:
  app: controlling
```

copy-lb.yaml

As noted earlier the load balancer for the copy instances does not support sessions. However you can use the load balancer to issue queries or simple inserts that don't require a session.

```
apiVersion: v1
kind: Service
metadata:
  name: copy-loadbalancer
spec:
  type: LoadBalancer
  ports:
    - port: 10035
      targetPort: 10035
  selector:
    app: copy
```

copy-0-lb.yaml

If you wish to access one of the copy instances explicitly so that you can create sessions you can create a load balancer which links to just one instance, in this case the first copy instance which is named "copy-0".

```
apiVersion: v1
kind: Service
metadata:
  name: copy-0-loadbalancer
spec:
  type: LoadBalancer
  ports:
    - port: 10035
      targetPort: 10035
  selector:
    app: copy
    statefulset.kubernetes.io/pod-name: copy-0
```

Setting up and running MMR under Kubernetes

The code will build and deploy an AllegroGraph MMR cluster in Kubernetes. We've tested this in Google Cloud Platform and Amazon Web Service. This code requires Persistent Volumes and load balancers and thus requires a sophisticated platform to run (such as GCP or AWS).

Prerequisites

In order to use the code supplied you'll need two additional things

1. A Docker Hub account (<https://hub.docker.com>). A free account will work. You'll want to make sure you can push to the hub without needing a password (use the docker login command to set that up).
2. An AllegroGraph distribution in tar.gz format. We've been using `agraph-6.6.0-linuxamd64.64.tar.gz` in our testing. You can find the current set of server files at <https://franz.com/agraph/downloads/server> This file should be put in the `ag` subdirectory. Note that the Dockerfile in that directory has the line `ARG agversion=agraph-6.6.0` which specifies the version of agraph to install. This must match the version of the `...tar.gz` file you put in that directory.

Steps

Do Prerequisites

Fullfill the prerequisites above

Set parameters

There are 5 parameters

1. Docker account – **Must Specify**
2. AllegroGraph user – **May want to specify**
3. AllegroGraph password – **May want to specify**
4. AllegroGraph repository name – **Unlikely to want to change**

5. AllegroGraph port – **Very unlikely to want to change**

The first three parameters can be set using the Makefile in the top level directory. The last two parameters are found in `agrep/vars.sh` if you wish to change them. Note that the port number of 10035 is found in the `yaml` files in the `kube` subdirectory. If you change the port number you'll have edit the `yaml` files as well.

The first three parameters are set via

```
% make account=DockerHubAccount user=username  
password=password
```

The account must be specified but the last two can be omitted and default to an AllegroGraph account name of *test* and a password of *xyzyz*.

If you choose to specify a password make it a simple one consisting of letters and numbers. The password will appear in shell commands and URLs and our simple scripts don't escape characters that have a special meaning to the shell or URLs.

Install AllegroGraph

Change to the `ag` directory and build an image with AllegroGraph installed. Then push it to the Docker Hub

```
% cd ag  
% make build  
% make push  
% cd ..
```

Create cluster-aware AllegroGraph image

Add scripts to create an image that will either create an AllegroGraph MMR cluster or join a cluster when started.

```
% cd agrepl  
% make build  
% make push
```

```
% cd ..
```

Setup a Kubernetes cluster

Now everything is ready to run in a Kubernetes cluster. You may already have a Kubernetes cluster running or you may need to create one. Both Google Cloud Platform and AWS have ways of creating a cluster using a web UI or a shell command. When you've got your cluster running you can do

```
% kubectl get nodes
```

and you'll see your nodes listed. Once this works you can move into the next step.

Run an AllegroGraph MMR cluster

Starting the MMR cluster involves setting up a number of services and deploying pods. The Makefile will do that for you.

```
% cd kube  
% make doall
```

You'll see when it displays the services that there isn't an external IP address allocated for the load balancers. It can take a few minutes for an external IP address to be allocated and the load balancers setup so keep running

```
% kubectl get svc
```

until you see an IP address given, and even then it may not work for a minute or two after that for the connection to be made.

Verify that the MMR cluster is running

You can use AllegroGraph Webview to see if the MMR cluster is running. Once you have an external IP address for the controlling-load-balancer go to this address in a web browser

`http://external-ip-address:10035`

Login with the credentials you used when you created the Docker images (the default is user *test* and password *xyzzzy*). You'll see a repository *myrepl* listed. Click on that. Midway down you'll see a link titled

Manage Replication Instances as controller

Click on that link and you'll see a table of three instances which now serve the same repository. This verifies that three pods started up and all linked to each other.

Namespaces

All objects created in Kubernetes have a name that is chosen either by the user or Kubernetes based on a name given by the user. Most names have an associated namespace. The combination of namespace and name must be unique among all objects in a Kubernetes cluster. The reason for having a namespace is that it prevents name clashes between multiple projects running in the same cluster that both choose to use the same name for an object.

The default namespace is named *default*.

Another big advantage using namespaces is that if you delete a namespace you delete all objects whose name is in that namespace. This is useful because a project in Kubernetes uses a lot of different types of objects and if you want to delete all the objects you've added to a Kubernetes cluster it can take a while to find all the objects by type and then delete them. However if you put all the objects in one namespace then you need only delete the namespace and you're done.

In the Makefile we have this line

`Namespace=testns`

which is used by this rule

reset:

```
-kubectl delete namespace ${Namespace}  
kubectl create namespace ${Namespace}  
    kubectl config set-context `kubectl config current-  
context` --namespace ${Namespace}
```

The reset rule deletes all members of the Namespace named at the top of the Makefile (here testns) and then recreates the namespace and switches to it as the active namespace. After doing the reset all objects created will be created in the testns namespace.

We include this in the Makefile because you may find it useful.

Docker Swarm

The focus of this document is Kubernetes but we also have a Docker Swarm implementation of an AllegroGraph MMR cluster. Docker Swarm is significantly simpler to setup and manage than Kubernetes but has far fewer bells and whistles. Once you've gotten the ag and agrepl images built and pushed to the Docker Hub you need only link a set of machines running Docker together into a Docker Swarm and then

```
% cd swarm ; make controlling copy
```

and the AllegroGraph MMR cluster is running Once it is running you can access the cluster using Webview at

<http://localhost:10035/>

Adding Properties to Triples

in AllegroGraph

AllegroGraph provides two ways to add metadata to triples. The first one is very similar to what typical property graph databases provide: we use the named graph of triples to store meta data about that triple. The second approach is what we have termed *triple attributes*. An attribute is a key/value pair associated with an individual triple. Each triple can have any number of attributes. This approach, which is built into AllegroGraph's storage layer, is especially handy for security and bookkeeping purposes. Most of this article will discuss triple attributes but first we quickly discuss the named graph (i.e. fourth element or quad) approach.

1.0 The Named Graph for Properties

Semantic Graph Databases are actually defined by the W3C standard to store RDF as 'Quads' (Named Graph, Subject, Predicate, and Object). The 'Triple Store' terminology has stuck even though the industry has moved on to storing quads. We believe using the named graph approach to store metadata about triples is richer model than the property graph database method.

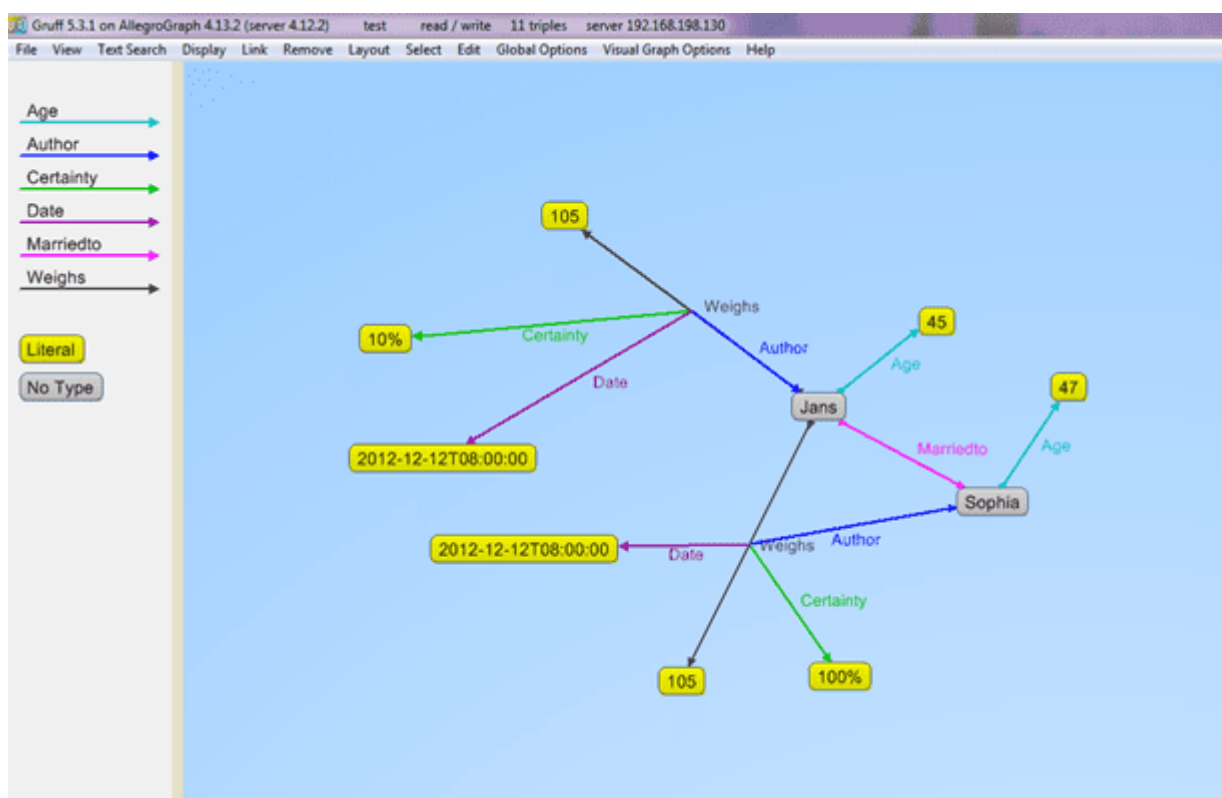
The best way to understand this is to give an example. Below we see two statements about Bruce weighing 105 kilos. The triple portions (subject, predicate, object) are identical but the named graphs (fourth elements) differ. They are used to provide additional information about the triples. The graph values are S1 and S2. By looking at these graphs we see that

- The author of the first triple (with graph S1) is Sophia and the author of the second (with graph S2) is Bruce (who is also the subject of the two triples).
- Sophia is 100% certain about her statement while Bruce is only 10% certain about his.

Using the named graph we can do even more than a property graph database, as the value of a graph can itself be a node, and is the subject of various triples which specify the original triple's author, date, and certainty. Additional triples tell us the ages of the authors and the fact that the authors are married.

Subject	Predicate	Object	Graph
Bruce	Weighs	105kg	S1
Bruce	Weighs	105kg	S2
Bruce	Age	45	None
Bruce	<u>MarriedTo</u>	Sophia	None
S1	Author	Sophia	None
S1	Date	2012-12-12T08:00:00	None
S1	Certainty	100 %	None
S2	Author	Bruce	None
S2	Date	2012-12-12T08:00:00	None
S2	certainty	10%	None
Sophia	Age	47	None

Here is the data displayed in Gruff, AllegroGraph's associated triple store browser:



Using named graphs for a triple's metadata is a powerful tool but it does have limitations: (1) only one graph value can be associated with a triple, (2) it can be important that metadata is stored directly and physically with the triple (with named graphs, the actual metadata is usually stored in additional triples with the graph as the subject, as in the example above), and (3) named graphs have competing uses and may not be available for metadata.

2.0 The Triple Attributes approach

AllegroGraph uniquely offers a mechanism called *triple attributes* where a collection of user defined key/value pairs can be stored with each individual triple. The advantage of this approach is manifold, but the original use case was designed for triple level security for an Intelligence agency.

By having triple attributes physically connected to the triples in the storage layer we can provide a very powerful and flexible mechanism to protect triples at the lowest possible level in AllegroGraph's architecture. Our first example below shows this use case in great detail. Other use cases are for example to add weights or costs to triples, to be used in graph algorithms. Or we can add a recorded time or expiration times to a triple and use that to provide a time machine in AllegroGraph or do automatic clean-up of old data.

Example with Attributes:

Subject – <http://dbpedia.org/resource/Arif_Babayev>

Predicate – <<http://dbpedia.org/property/placeOfDeath>>

Object – <<http://dbpedia.org/resource/Baku>>

Named Graph – <<http://ex#trans@@1142684573200001>>

Triple Attributes – {"securityLevel": "high",

```
"department": "hr", "accessToken": ["E", "D"]}
```

This article provides an initial introduction to attributes and the associated concept of static filters, showing how they are set up and used. We start with a security example which also describes the basics of adding attributes to triples and filtering query results based on attribute values. Then we discuss other potential uses of attributes.

2.1 Triple Attribute Basics: a Security Example

One important purpose of attributes, when they were added as a feature, was to allow for very fine triple-level security, so that triples would be visible or invisible to users according to the attributes of the triples and the permissions associated with the query being posed by the user.

Note that users as such do not have attributes. Instead, attribute values are assigned when a query is posed. This is an important point: it is natural to think that there can be an attribute SECURITY-LEVEL, and a triple can have attribute SECURITY-LEVEL=3, and USER1 can have an attribute SECURITY-LEVEL=2 and USER2 can have an attribute SECURITY-LEVEL=4, and the system can require that the user SECURITY-LEVEL attribute must be greater than the triple SECURITY-LEVEL for the triple to be visible to the user. But that is not how attributes work. The triples can have the attribute SECURITY-LEVEL=2 but users do not have attributes. Instead, the filter is made part of the query.

Here is a simple example. We define attributes and static attribute filters using AGWebView. We have a repository named repo. Here is a portion of its AGWebView page:

Repository repo — 0 statements

[\[edit description\]](#)

Load and Delete Data

- [Add a statement](#)
- [Delete statements](#)
- [Import RDF:](#)
 - [from an uploaded file](#)
 - [from a server-side file](#)
 - [from a text area input](#)

Explore the Repository

- [View triples](#)
- [View quads](#)
- [View repository's classes](#)
- [View repository's predicates](#)
- [View repository's named graphs](#)

Reports

- [Storage report](#)
- [Triple indices](#)
- [String table](#)
- [Full list of reports ...](#)


Multi-Master Replication

- [Convert store to a replication instance](#)

Warm Standby Replication

- [Control replication](#)

Repository Control

- [Export repository as](#)
- [Start a session](#) — support transactions and Prolog functors
- [Warmup store](#)
- [Back-up this repository](#)
- [Export duplicate statements](#)
- [Delete duplicate statements](#)
- [Suppress duplicate statements](#)
- [View active transactions](#)
- [Recognize geospatial datatypes automatically:](#) ☐
- [Control durability \(bulk-load mode\)](#)
- [Manage attribute definitions](#)
- [Set static attribute filter](#)
- [Manage triple indices](#)
 - [Optimize the repository](#) 



The red arrow points to the commands of interest: Manage attribute definitions and Set static attribute filter. We click on Set static attribute filter to define an attribute. We have filled in the attribute information (name *security-level*, minimum and maximum number allowed per triple, allowed values, and whether order or not (yes in our case):

AllegroGraph WebView

repository repo

[Repository](#) | [Queries](#) | [Utilities](#) | [Admin](#) | [User test](#)

New Attribute

Name

security-level

Min. number

0

Max. number

1

Allowed Values

0,1,2,3,4,5

Ordered

☒

Save

Cancel

Attribute Definitions

+ Add New

Name	Min. number	Max. number	Allowed values	Ordered	
No attributes have been defined					

We click Save and the attribute is defined:

AllegroGraph WebView

repository repo

[Repository](#) | [Queries](#) | [Utilities](#) | [Admin](#) | [User test](#)

Attribute Definitions

+ Add New

Name	Min. number	Max. number	Allowed values	Ordered	
security-level		1	0,1,2,3,4,5	✓	

Then we define a filter (on the Set static attribute filter page) :

AllegroGraph WebView

repository repo

[Repository](#) | [Queries](#) | [Utilities](#) | [Admin](#) | [User test](#)

Static Filter

Current filter

(attribute-set> user.security-level triple.security-level)

Edit filter

(attribute-set> user.security-level triple.security-level)

Save

Revert to current

Clear

We defined the filter `(attribute-set> user.security-level triple.security-level)` and clicked Save (the definition appears in both the Edit and the Current fields). The filter says that the “user” security level must be greater than the triple security level. We put “user” in quotes because the user security level is specified as part of the query, and has no direct connection to any specific user.

Here are some triples in a nqx file *fr.nqx*. The first triple has no attributes and the other three each has a security-level attribute value.

<<http://www.franz.com#emp0>>

```
<http://www.franz.com#position> "intern" .
```

```
<http://www.franz.com#emp1>
```

```
<http://www.franz.com#position> "worker" {"security-level":  
"2"} .
```

```
<http://www.franz.com#emp2>
```

```
<http://www.franz.com#position> "manager" {"security-level":  
"3"} .
```

```
<http://www.franz.com#emp3>
```

```
<http://www.franz.com#position> "boss" {"security-level": "4"}  
.
```

We load this file into a repository which has the security-level attribute defined as above and the static filter mentioned above also defined. (Triples with attributes can also be entered directly when using AGWebView with the Import RDF from a text area input command).

Once the triples are loaded, we click View triples in AGWebView and we see no triples:

AllegroGraph WebView repository repo

[^](#) | [Repository](#) | [Queries](#) | [Utilities](#) | [Admin](#) | [User test](#)

Edit query

1 |# View triples

2 SELECT ?s ?p ?o { ?s ?p ?o . }

Execute

Log Query

Show Plan

Save as

Add to repository

No results

This result is often surprising to users just beginning to work with attributes and filters, who may expect the first triple, abbreviated to [emp0 position intern], to be visible, but the system is doing what it is supposed to do. It will only show triples where the security-level of the user posing the query is greater than the security level of the triple. The user has no security level and so the comparison fails, even with triples that have no security-level attribute value. We will describe below how to ensure you can see triples with no attributes.

So we need to specify an attribute value to the user posing the query. (As said above, users do not themselves have attribute values. But the attribute value of a user posing a query can be specified as part of the query.) “User” attributes are specified with a prefix like the following:

prefix franzOption_userAttributes: <franz:%7B%22security-

```
level%22%3A%223%22%7D>
```

so the query should be

```
prefix franzOption_userAttributes: <franz:%7B%22security-  
level%22%3A%223%22%7D>
```

```
select ?s ?p ?o { ?s ?p ?o . }
```

We will show the results below, but first what are all the % signs and numbers doing there? Why isn't the prefix just `prefix franzOption_userAttributes: <franz:{"security-level":"3"}>`? The issue is that `{"security-level":"3"}` won't read correctly. It must be URL encoded. We do this by going to <https://www.urlencoder.org/> (there are other websites that do this as well) and put `{"security-level":"3"}` in the first box, click Encode and get `%7B%22security-level%22%3A%223%22%7D`. We then paste that into the query, as shown above.

When we try that query in AGWebView, we get one result:

AllegroGraph WebView

repository repo

[Repository](#) | [Queries](#) | [Utilities](#) | [Admin](#) | [User test](#)

Edit query

```

1 prefix franzOption_userAttributes: <franz:%7B%22security-level%22%3A%225%22%7D>
2 select ?s ?p ?o { ?s ?p ?o . }
3

```

Execute

Log Query

Show Plan

Save as

Add to repository

1 Result in 2.619 ms

Information

s	p	o
<http://www.franz.com#emp1>	<http://www.franz.com#position>	"worker"

If we encode {"security-level": "5"} to get the query

```

prefix  franzOption_userAttributes:  <franz:%7B%22security-
level%22%3A%225%22%7D>
select ?s ?p ?o { ?s ?p ?o . }

```

we get three results:

emp3	position	"boss"
emp2	position	"manager"
emp1	position	"worker"

since now the "user" security-level is greater than that of any triples with a security-level attribute. But what about

the triple with subject emp0, the triple with no attributes? It does not pass the filter which required that the user attribute be greater than the triple attribute. Since the triple has no attribute value so the comparison failed.

Let us redefine the filter to:

```
(or (attribute-set> user.security-level triple.security-level)
    (empty triple.security-level))
```

The screenshot shows the AllegroGraph WebView interface. At the top, there is a header bar with the title "AllegroGraph WebView" and a sub-header "repository repo". Below this is a navigation bar with links: "Repository", "Queries", "Utilities", "Admin", and "User test". The main content area is titled "Static Filter". It contains two sections: "Current filter" and "Edit filter". Both sections display the same filter expression:

```
(or (attribute-set> user.security-level triple.security-level)
    (empty triple.security-level))
```

. At the bottom of the "Edit filter" section, there are three buttons: "Save", "Revert to current", and "Clear".

Now a triple will pass the filter if either (1) the “user” security-level is greater than the triple security-level or

(2) the triple does not have a security-level attribute. Now the query from above where the user has attribute security-level:"5" will show all the triples with security-level less than 5 and with no attributes at all. That happens to be all four triples so far defined:

AllegroGraph WebView repository repo

Repository | Queries | Utilities | Admin | User test

Edit query

```
1 prefix franzOption_userAttributes: <franz:%7B%22security-level%22%3A%225%22%7D>
2 select ?s ?p ?o { ?s ?p ?o . }
3
```

Execute Log Query Show Plan Save as Add to repository

4 Results in 0.286 ms Information

s	p	o
<http://www.franz.com#emp0>	<http://www.franz.com#position>	"intern"
<http://www.franz.com#emp3>	<http://www.franz.com#position>	"boss"
<http://www.franz.com#emp2>	<http://www.franz.com#position>	"manager"
<http://www.franz.com#emp1>	<http://www.franz.com#position>	"worker"

The triple

emp0 position "intern"

will now appears as a result in any query where it satisfies the SPARQL select regardless of the security-level of the

“user”.

It would be a useful feature that we could associate attributes with actual users. However, this is not as simple as it sounds. Attributes are features of repositories. If I have a REP01 repository, it can have a bunch of defined attributes and filters but my REP02 may know nothing about them and its triples may not have any attributes at all, and no attributes are defined, and (as a result) no filters. But users are not repository-linked objects. While a repository can be made read-only or unreadable for a user, users do not have finer repository features. So an interface for providing users with attributes, since it would only make sense on a per-repository basis, requires a complicated interface. That is not yet implemented (though we are considering how it can be done).

Instead, users can have specific prefixes associated with them and that prefix and be included in any query made by the user.

But if all it takes to specify “user” attributes is to put the right line at the top of your SPARQL query, that does not seem to provide much security. There is a feature for users “Allow user attributes via SPARQL PREFIX `franzOption_userAttributes`” which can restrict a user’s ability to specify “user” attributes in a query, but that is a rather blunt instrument. Instead, the model is that most users (outside of trusted administrators) are not actually allowed to pose SPARQL queries directly. Instead, there is an intermediary program which takes the query a user requests and, having determined the status of the user and what attribute values should be given to the user, modifies the query with the appropriate `franzOption_userAttributes` prefixes and then sends the query on to the server, following which it captures the results and

sends them back to the requesting user. That intermediate program will store the prefix suitable for a user and thus associate “user” attributes with specific users.

2.2 Using attributes as additional data

Although triple security is one powerful use of attributes, security is far from the only use. Just as the named graph can serve as additional data, so can attributes. SPARQL queries can use attribute values just as static filters can filter out triples before displaying them. Let us take a simple example: the attribute `timeAdded`. Every triple we add will have a `timeAdded` attribute value which will be a string whose contents are a datetime value, such as “2017-09-11T:15:52”. We define the attribute:

Attribute Definitions

+ Add New					
Name	Min. number	Max. number	Allowed values	Ordered	
timeAdded	1	1			

Now let us define some triples:

```
<http://www.franz.com#emp0>
<http://www.franz.com#callRank> "2" {"timeAdded":
"2019-01-12T10:12:45" } .
<http://www.franz.com#emp0>
<http://www.franz.com#callRank> "1" {"timeAdded":
"2019-01-14T14:16:12" } .
<http://www.franz.com#emp0>
<http://www.franz.com#callRank> "3" {"timeAdded":
"2019-01-11T11:15:52" } .
```

```

    <http://www.franz.com#emp1>
<http://www.franz.com#callRank> "5" {"timeAdded":
"2019-01-13T11:03:22" } .
    <http://www.franz.com#emp0>
<http://www.franz.com#callRank> "2" {"timeAdded":
"2019-01-13T09:03:22" } .

```

We have a call center with employees making calls. Each call has a ranking from 1 to 5, with 1 the lowest and 5 the highest. We have data on five calls, four from emp0 and one from emp1. Each triples has a timeAdded attribute with a string containing a dateTime value. We load these into a empty repository named at-test where the timeAdded attribute is defined as above:



SPARQL queries can use the attribute magic properties (see <https://franz.com/agraph/support/documentation/current/triple-attributes.html#Querying-Attributes-using-SPARQL>). We use the attributesNameValue magic property to see the subject, object, and attribute value:

```

select ?s ?o ?value {
                                (?ta      ?value)

```

```

<http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>
  (?s ?p ?o) .
}

```

Repository | Queries | Utilities | Admin | User test

Edit query

```

1 select ?s ?o ?value {
2   (?ta ?value) <http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>   (?s ?p ?o) .
3 }

```

Execute Log Query Show Plan Save as Add to repository

5 Results in 0.363 ms Information

s	o	value
<http://www.franz.com#emp0>	"2"	"2019-01-13T09:03:22"
<http://www.franz.com#emp1>	"5"	"2019-01-13T11:03:22"
<http://www.franz.com#emp0>	"3"	"2019-01-11T11:15:52"
<http://www.franz.com#emp0>	"1"	"2019-01-14T14:16:12"
<http://www.franz.com#emp0>	"2"	"2019-01-12T10:12:45"

But we are really interested just in emp0 and we would like to see the results ordered by time, that is by the attribute value, so we restrict the query to emp0 as the subject and order the results:

```

select ?o ?value {
  (?ta ?value)
  <http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>
  (<http://www.franz.com#emp0> ?p ?o) .
} order by ?value

```

Edit query

```
1 select ?o ?value {  
2   (?ta ?value) <http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>    (<http://www.franz.com#emp0> ?p ?o) .  
3 } order by ?value
```

Execute

Log Query

Show Plan

Save as

Add to repository

4 Results in 0.630 ms

Information

o	value
"3"	"2019-01-11T11:15:52"
"2"	"2019-01-12T10:12:45"
"2"	"2019-01-13T09:03:22"
"1"	"2019-01-14T14:16:12"

There are the results for emp0, who is clearly having difficulties because the call rankings have been steadily falling over time.

Another example using timeAdded is employee salary data. In the Human Resources data, the salary of an employee is stored:

emp0 hasSalary 50000

Now emp0 gets a raise to 55000. So we delete the triple above and add the triple

emp0 hasSalary 55000

But that is not satisfactory because we have lost the salary

history. If the boss asks “How much was emp0 paid initially?” we cannot answer. There are various solutions. We could define a salary change object, with predicates effectiveDate, previousSalary, newSalary, and so on:

```
salaryChange017 forEmployee emp0
salaryChange017 effectiveDate "2019-01-12T10:12:45"
salaryChange017 oldSalary "50000"
salaryChange017 newSalary "55000"
```

emp0 hasSalaryChange salaryChange017

and that would work fine, but perhaps it is more setup and effort than is needed. Suppose we just have `hasSalary` triples each with a `timeAdded` attribute. Then the current salary is the latest one and the history is the ordered list. Here that idea is worked out:

```
<http://www.franz.com#emp0>    <http://www.franz.com#hasSalary>
"50000"^^<http://www.w3.org/2001/XMLSchema#integer>
{"timeAdded": "2017-01-12T10:12:45" } .

<http://www.franz.com#emp0>    <http://www.franz.com#hasSalary>
"55000"^^<http://www.w3.org/2001/XMLSchema#integer>
{"timeAdded": "2019-03-17T12:00:00" } .
```

What is the current salary? A simple SPARQL query tells us:

```

select ?o ?value {
                                (?ta      ?value)
<http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>
                                (<http://www.franz.com#emp0>
<http://www.franz.com#hasSalary> ?o) .
    } order by desc(?value) limit 1

```

Edit query

```
1 select ?o ?value {  
2   (?ta ?value) <http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>  
3   (<http://www.franz.com#emp0> <http://www.franz.com#hasSalary> ?o) .  
4 } order by desc(?value) limit 1
```

Execute

Log Query

Show Plan

Save as

Add to repository

1 Result in 0.388 ms

Information

o	value
"55000"	"2019-03-17T12:00:00"

The salary history is provided by the same query without the LIMIT:

```
select ?o ?value {  
    (?ta ?value)  
<http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>  
    (<http://www.franz.com#emp0>  
<http://www.franz.com#hasSalary> ?o) .  
    } order by desc(?value)
```

Edit query

```
1 select ?o ?value {  
2   (?ta ?value) <http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>  
3   (<http://www.franz.com#emp0> <http://www.franz.com#hasSalary> ?o) .  
4 } order by desc(?value)
```

Execute

Log Query

Show Plan

Save as

Add to repository

2 Results in 0.413 ms

Information

o	value
"55000"	"2019-03-17T12:00:00"
"50000"	"2017-01-12T10:12:45"

This method of storing salary data may not easily support more complex questions which might be easily answered if we went the salaryChange object route mentioned above but if you are not looking to ask those questions, you should not do the extra work (and the risk of data errors) required.

You could use the graph of each triple for the timeAdded. All the examples above would work with minor tweaks. But there are many uses for the named graph of a triple. Attributes are available and using them for one purpose does not restrict their use for other purposes.

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>