

# **AllegroGraph – KMWorld Readers Choice Finalist**

**KMWorld 2019 Readers' Choice Awards: Best Knowledge Graph**

## **AllegroGraph – Finalist**

The ability for knowledge graphs to amass information and relationships and connect those facts allows companies to find context in data, which is important for extracting value as well as complying with new data regulations.

The concept of the enterprise knowledge graph (EKG) is fairly new and made possible by machine learning and big data technologies, including automated text analysis and graph engines, explained analyst Amy Stapleton in an Opus Research article. "An IA [intelligent assistant] that taps into an EKG can infer the context and intent of questions, generate direct answers, make recommendations, and automatically expand its understanding as the knowledge graph adds new content," she noted.

KMWorld Readers Choice

---

## **Three Necessities For Maximizing Your Digital Twins Approach**

The digital twin premise is arguably the most viable means of implementing equipment asset management throughout the industrial internet. It's an exceptionally lucrative element

of the internet of things (IoT), with an applicability that easily lends itself to numerous businesses. Its real-time streaming data, simulation capabilities and relationship awareness may well prove to be the “killer app” that makes the IoT mainstream.

## **Digital Twins Types**

There are presently three types of digital twins: those for individual assets, operations and predictions. In this article, we will focus on individual assets. Examples of these assets include drilling machines in the oil and gas industry or assembly line equipment. Each type of digital twin creates a three-dimensional simulation of the real-world features it models based on relationships of IoT data. The simulated models capture and contextualize this low-latent data about each asset for vital visibility into its performance. This real-time data provides a blueprint for diminishing downtime, scheduling maintenance and monitoring other factors that impact overall asset productivity and ROI. At scale, each factor translates into significant savings, increased performance and greater chances for optimization.

The crux of the digital twin’s expansive capabilities is almost entirely predicated on solving one of the more time-honored data management difficulties: data modeling. But the schema issues complicating downstream data modeling processes such as transformation, integration and predictive analytics can be swiftly redressed by knowledge graphs that simplify this vital prerequisite. The standards-based data models of semantic knowledge graphs deliver unparalleled flexibility, interoperability and low latency for which IoT deployments are renowned. *(Full disclosure: My company specializes in semantic knowledge graphs.)*

Read the Full Article at Forbes.

---

# 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 `agrep1/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/>

---



# Graphorum — Dr. Aasman Presenting

## Graph-Driven Event Processing for Intelligent Customer Operations

Wednesday, October 16, 2019

10:15 AM – 11:15 AM

Level: Case Study



Dr. Jans Aasman, CEO, Franz Inc.

In the typical organization, the contents of the actual chat or voice conversation between agent and customer is a black hole. In the modern Intelligent Customer Operations center, the interactions between agent and customer are a source of rich information that helps agents to improve the quality

of the interaction in real time, creates more sales, and provides far better analytics for management. The Intelligent Customer Operations center is enabled by a taxonomy of the products and services sold, speech recognition to turn conversations into text, a taxonomy-driven entity extractor to take the important concepts out of conversations, and machine learning to classify chats in various ways. All of this is stored in a real-time Knowledge Graph that also knows (and stores) everything about customers and agents and provides the raw data for machine learning to improve the agent/customer interaction.

In this presentation, we describe a real-world Intelligent Customer Organization that uses graph-based technology for taxonomy-driven entity extraction, speech recognition, machine learning, and predictive analytics to improve quality of conversations, increase sales, and improve business

visibility.

<https://graphorum2019.dataversity.net/sessionPop.cfm?confid=132&proposalid=11010>

---

# Fixed Indices Speed up Slot Access in Allegro CL

Instances of CLOS classes include a vector of slot values associated with the instance. A new feature in Allegro CL allows these slot values to be accessed more efficiently by specifying at **defclass** time the index of the slot values vector a particular slot will occupy.

## 1.0 Background

Object-oriented programming and specifically CLOS (the Common Lisp Object System) provide a powerful programming tool but it comes at a cost: the more complex the hierarchy of classes and the provision of methods, the longer it takes for the system to determine exactly what should be done in a specific method call. When a method is called, the class of every required argument must be determined and all methods associated with the class pattern of the arguments must be found.

One common action is determining the value of a slot of a class instance. Slots are specified when a CLOS class is defined, with some slots being inherited from superclasses and some being added by the new class definition. Regardless of whether slots are inherited or new, when an instance of the class is created, a vector of slot values is associated with

the instance.

Thus to obtain an instance's slot value, the system must only determine the index of the particular slot in the vector of values. At its least optimized, finding the index involves accessing information in the class definition, which can be costly. This access is what takes time when getting a slot value: once the index is known, the value is obtained by a simple vector access.

## 2.0 Fixed indices

It is possible that the structure of the slot value vector (that is, which index corresponds to which slot) might change if a class is redefined. Therefore it is unsafe to assume that, say, if a slot value is at index 2 in one call it will be at that index in a later call. Slot accesses cannot, using standard Common Lisp techniques be fully optimized.

Allegro CL has added a new feature which improves upon this: fixed index values. Now when a class is defined, each slot can be assigned an index value and the system guarantees that the slot value will always be at that index of the slot value vector. This means that a slot value access can be reduced to

```
(svref [slot-value-vector] [slot-index])
```

which, with proper declarations and optimization levels, can be reduced to a few machine instructions. This can provide significant speed-ups, as the following example shows.

```
(defclass foo ())  
(a :initarg :a excl:fixed-index 2 :accessor foo-a)  
(b :initarg :b excl:fixed-index 3 :accessor foo-b)  
(c :initarg :c :accessor foo-c)))  
(defvar *foo-inst* (make-instance 'foo :a 1 :b 2 :c 3))
```

```
;; *VEC* is the slot values vector for *F00-INST*.  
;; The value of the 'A slot is index 2.
```

```

(defvar *vec* (excl:std-instance-slots *foo-inst*))

;; The slot value vector of *foo-inst* has four elements
for three
;; values:
;; 0 – value if slot c
;; 1 – unused
;; 2 – value of slot a
;; 3 – value of slot b

;; Let's time some accesses. We define test functions in a
file:

;; file sv.cl
(in-package :user)

(defun p1 () (dotimes (i 10000000) (foo-a *foo-inst*)))
(defun p2 () (dotimes (i 10000000) (slot-value *foo-inst*
'a)))
(defun p3 () (dotimes (i 10000000) (svref *vec* 2)))
;; sv.cl end

;; We compile with speed 3:
Compiler optimize setting is
(declaim (optimize (safety 1) (space 1) (speed 3) (debug 0)
(compilation-speed 0)))
cl-user(39): :cl sv.cl
;;; Compiling file sv.cl
;;; Writing fasl file sv.fasl
;;; Fasl write complete
; Fast loading sv.fasl

;; And we run timings:
cl-user(41): (time (p1))  ;; USING THE F00-A ACCESSOR:
; cpu time (non-gc) 0.127957 sec user, 0.000000 sec system
; cpu time (gc)      0.000000 sec user, 0.000000 sec system
; cpu time (total) 0.127957 sec user, 0.000000 sec system

```

```

; real time 0.127954 sec (100.0%)
; space allocation:
; 0 cons cells, 0 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 88 (gc: 0)
nil
cl-user(42): (time (p2)) ;; USING SLOT-VALUE:
; cpu time (non-gc) 0.274489 sec user, 0.000000 sec system
; cpu time (gc)      0.000000 sec user, 0.000000 sec system
; cpu time (total) 0.274489 sec user, 0.000000 sec system
; real time 0.274485 sec (100.0%)
; space allocation:
; 0 cons cells, 0 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
nil
cl-user(43): (time (p3)) ;; USING SVREF ON THE SLOT VALUE
VECTOR:
; cpu time (non-gc) 0.005561 sec user, 0.000994 sec system
; cpu time (gc)      0.000000 sec user, 0.000000 sec system
; cpu time (total) 0.005561 sec user, 0.000994 sec system
; real time 0.006555 sec (100.0%)
; space allocation:
; 0 cons cells, 0 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)

```

Accesses using **svref** is thus 50 times faster than ordinary access.

You will note we specified the value vector index of a slot using the slot option `excl:fixed-index` in the **defclass** form. This is an Allegro CL extension and if used in code which will also be run in other Common Lisp implementation must be conditionalized for Allegro CL only.

## 2.1 Before, after, and around methods

In the example above, we reduced a slot access to a **svref** call into the slot value vector. This will produce the maximum speedup but users must keep in mind that before, around, and

after methods defined on **slot-value** or on a slot accessor function (**foo-a** and **foo-b** in our example as slot **c** did not have a fixed index) will not be run in that case because the actual methods are not run, just the call to **svref**. Users who intend to write before, around, or after methods on slot accessors can get the desired behavior and much of the speedup with one additional level of indirection:

```
(defmethod generic-foo-a ((obj foo)) (foo-a obj))
```

Before, after, and around methods can then be written for **generic-foo-a** and will run as expected.

## 3.0 Documentation

Allegro CL documentation can be found at <https://franz.com/support/documentation/>. Fixed-index slots are described in the section Defclass optimizations: fixed-index slots and defclass embellishers in the implementation.htm document. Also described in that section are **defclass embellishers**, which we will discuss in a later blog item.

---

# Big Data 50 – Companies Driving Innovation in 2019

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



Today, more than ever, businesses rely on data to deliver a competitive edge. The urgency to compete on analytics has spread across industries, fueled

by the need for greater efficiency, agility and innovation,” remarked Thomas Hogan, Group Publisher at Database Trends and Applications. “This list seeks to highlight those companies that are really driving innovation and serve as a guide to businesses navigating the rapidly changing big data landscape.”

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

But, it is also increasingly clear that there is no single way to approach data-driven innovation today. Open source-based technologies have gained strong adoption in organizations alongside proprietary offerings, data lakes are increasingly being implemented but data warehouses continue in widespread use, and hybrid deployments spanning cloud and on-premise are commonly accepted.

Organizations are seeking to use data-driven innovation for better reporting and analytics, real-time decision making, enhanced customer experience and personalization, and reduced costs. But with data coming in from more places than ever, being stored in more systems, and accessed by more users for a

wider array of use cases, there is greater recognition that security and governance must be addressed intelligently.

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

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

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

---

## **The Importance of FAIR Data in Earth Science**

**Franz’s CEO, Jans Aasman’s recent Marine Technology News:**

Data’s valuation as an enterprise asset is most acutely realized over time. When properly managed, the same dataset





supports a plurality of use cases, becomes almost instantly available upon request, and is exchangeable between departments or organizations to systematically increase its yield with each deployment.

These boons of leveraging data as an enterprise asset are the foundation of GO FAIR's Findable Accessible Interoperable Reusable (FAIR) principles profoundly impacting the data management rigors of geological science. Numerous organizations in this space have embraced these tenets to swiftly share information among a diversity of disciplines to safely guide the stewardship of the earth.

According to Dr. Annie Burgess, Lab Director of Earth Science Information Partners (ESIP), the "most pressing global challenges cannot be solved by a single organization. Scientists require data collected across multiple disciplines, which are often managed by many different agencies and institutions." As numerous members of the earth science community are realizing, the most effectual means of managing those disparate data according to FAIR principles is by utilizing the semantic standards underpinning knowledge graphs.

**Read the full article at [Marine Technology News](#)**

---

## **Ontolog Forum – Why Knowledge**

# Graphs Now (Recording)

Dr. Jans Aasman Presented

**Why Knowledge Graphs Hit the Hype Cycle and What they have in common.**

The Ontolog Forum is an open, international, virtual community of practice devoted to advancing the field of ontology, ontological engineering and semantic technology, and advocating their adoption into mainstream applications and international standards.

---

## Ontology Summit 2020 – Knowledge Graphs

The Ontology Summit is an annual series of events that involves the ontology community and communities related to each year's theme chosen for the summit. The Ontology Summit was started by Ontolog and NIST, and the program has been co-organized by Ontolog, NIST, NCOR, NCB0, IA0A, NCO\_NITRD along with the co-sponsorship of other organizations that are supportive of the Summit goals and objectives.

Knowledge graphs, closely related to ontologies and semantic networks, have emerged in the last few years to be an important semantic technology and research area. As structured representations of semantic knowledge that are stored in a graph, KGs are lightweight versions of semantic networks that scale to massive datasets such as the entire World Wide Web. Industry has devoted a great deal of effort to the development

of knowledge graphs, and they are now critical to the functions of intelligent virtual assistants such as Siri and Alexa. Some of the research communities where KGs are relevant are Ontologies, Big Data, Linked Data, Open Knowledge Network, Artificial Intelligence, Deep Learning, and many others.

Dr. Jans Aasman presented – ***“Why Knowledge Graphs Hit the Hype Cycle and What they have in common”***

Presentation Page

Presentation Slides



---

## **Harnessing the Internet of Things with JSON-LD**



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

Conceptually, the promise of the Internet of Things is almost halcyon. Its billions of sensors are all connected,

continuously transmitting data to support tailored, cost-saving measures maximizing revenues in applications as diverse as smart cities, smart price tags, and predictive maintenance in the Industrial Internet.

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

.....

The self-describing, linked data approach upon which JSON-LD is founded excels at the low latent action resulting from machine to machine communication in the IoT. The nucleus of the linked data methodology—semantic statements and their unique Uniform Resource Identifiers (URIs)—are read and understood by machines. This characteristic aids many of the IoT use cases requiring machine intelligence; by transmitting IoT data via the JSON-LD format organizations can maximize this boon. Smart cities provide particularly compelling examples of the machine intelligence fortified by this expression of semantic technology.

**Read the full article at [IoT Evolution](#)**