Bloor Research Covers AllegroGraph with FedShard

Written By: Daniel Howard
Published: 3rd June 2020

Link to Bloor Research Post on AllegroGraph 7

Graph database and *"knowledge graph solution"* vendor Franz Inc. has recently released its latest version of its graph offering, AllegroGraph, which as of the new release is in version 7.0. Alongside this, the company also released version 8.0 of Gruff, its visualisation and discovery engine for AllegroGraph and SPARQL endpoints. As of version 8, Gruff is now available via a web client as well as on the desktop.

For those of you who are unfamiliar with Franz and AllegroGraph, here is a brief description of the latter taken from our 2019 deep dive:

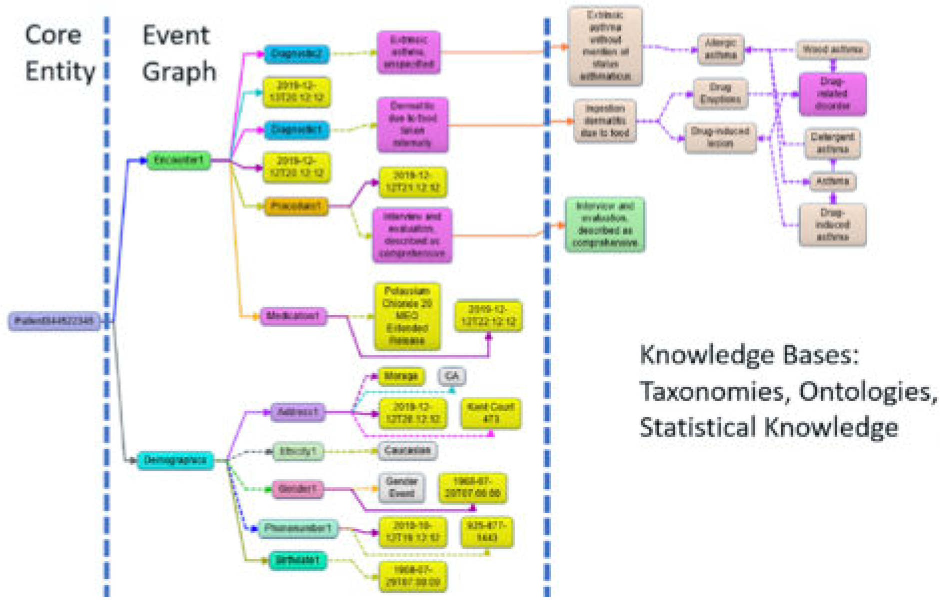
> *Franz AllegroGraph… is a semantic graph database focused on generating sophisticated semantic knowledge graphs, initially from your existing data. The graph database itself is an RDF-based quad store (in other words, a triple store where all the triples are named) with property graph support… The product's primary focus is on transactional processing; however, it is often used for analytics as well. Consequently, it is OLTP-enabled and fully ACID compliant, and additionally offers immediate consistency. The product is also highly secure, and supports the requirements for various*

> *government security standards, including HIPAA…It includes a wealth of features, including distributed deployment and querying, multi-modal ingestion, multi-master replication, AI and machine learning, and natural language processing (NLP).”*

The full text of this report can be found here.

As for the version 7.0 release of AllegroGraph, arguably the most compelling new capability is its ability to create what Franz refers to as *“Entity-Event Knowledge Graphs”* (or EEKGs) via its patented FedShard technology, an example which can be seen below. These differ from regular graphs in that they are designed to capture a number of core entities (such as products, patients or customers) and any events (which may be time-stamped) relating to those entities within a hierarchical tree structure. Notice in the image below that each node – each event – *“branches”* into one or more additional events, and that these branches usually do not interact: this is characteristic of the tree model. Moreover, these events can terminate in a knowledge base, such as a taxonomy or an ontology. This allows you to bring a wealth of supporting information into your graph when that information may be relevant to your core entity (or entities). For example, suppose you have a core entity that is a hospital patient. In this case, you might use a publicly available catalogue of drug interactions as one of your knowledge bases. This all creates an intuitive way to visually represent your entities and the events related to them while bringing in outside knowledge where it is useful. In this context it is also worth noting that AllegroGraph has probabilistic capabilities (this is not a new feature) so that, for example, a physician can assign a probability to a diagnosis.

Notably, your EEKGs can be built incrementally, starting with a simple model and extending gradually and as needed, but without requiring you to actually alter any existing parts of your model. EEKGs also store provenance information for your events, which captures where the initial data used to generate each of your events originally came from, and how its transformation into a graph object was achieved, thus providing data lineage.


Core Entity
Event Graph
Knowledge Bases: Taxonomies, Ontologies, Statistical Knowledge

The FedShard feature in the 7.0 release provides enhanced capabilities for horizontally distributed deployment, and EEKGs especially have been designed to take advantage of this. Other new features include improved JSON and JSON-LD document handling, Natural Language Processing (NLP) and speech recognition functionality. For the latter in particular, Franz has been able to leverage voice to text capabilities to extract conceptual meaning from real speech, then store that meaning in a graph and subsequently run analytics on it. This is exciting, because in effect it allows you to analyse recorded conversations. For organisations that collect and store a lot of such conversations – call centres, for instance – this could prove very useful.

Franz clearly considers this a major release for AllegroGraph. Certainly, the introduction of an explicit entity-event graph is not something I've seen before. The newly introduced text to speech capabilities also seem highly promising.

AllegroGraph Tutorial – Distributed Repository Using Shards and Federation Setup

Introduction

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

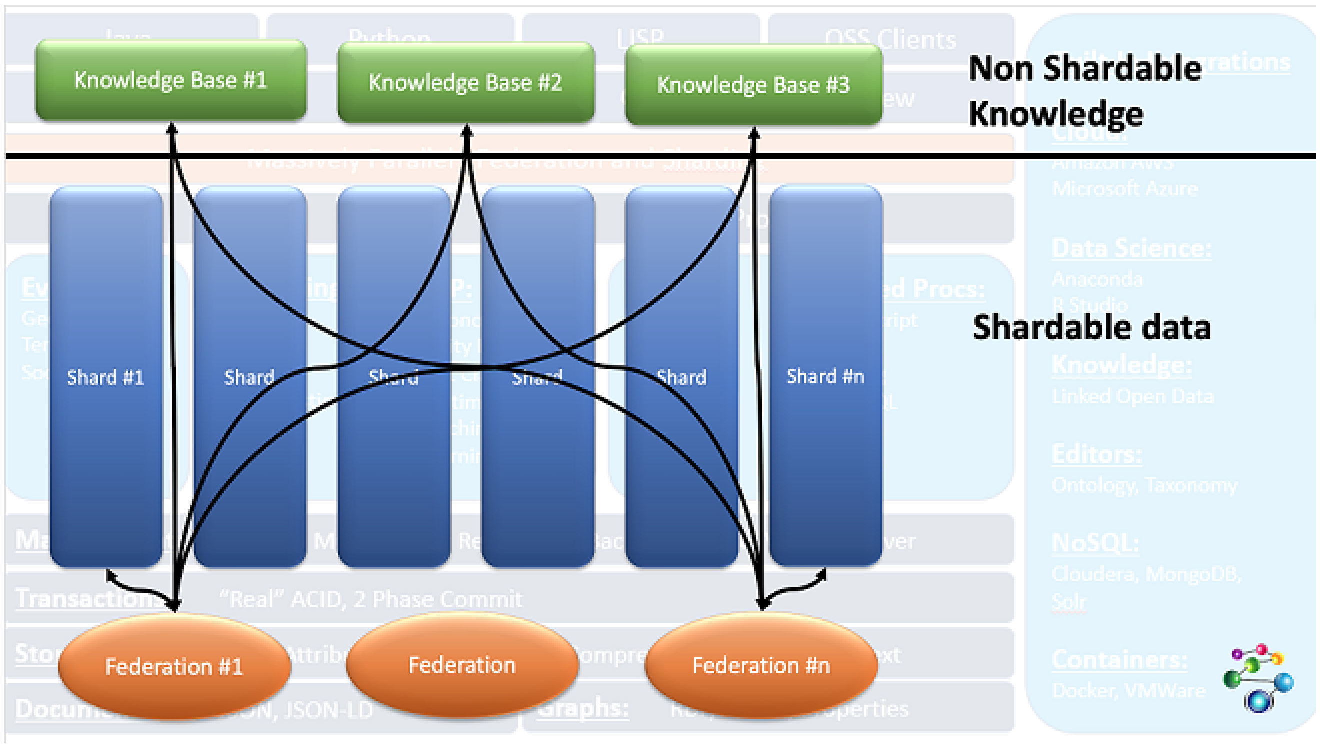
When the demands of data or query patterns outpace the ability of a server to keep up, there are two ways to attempt to grow your system: vertical or horizontal scaling.

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

AllegroGraph provides an alternative solution: a cluster of database servers, and horizontal scaling through *sharding* and *federation*, which combine in AllegroGraph's FedShard™ facility. An AllegroGraph cluster is a set of AllegroGraph installations across a defined set of machines. A distributed repository is a logical database comprised of one or more repositories spread across one or more of the AllegroGraph nodes in that cluster. A distributed repository has a required partition key that is used when importing statements. When adding statements to your repository, the partition key is used to determine on which shard each statement will be placed. By carefully choosing your partition key, it is possible to distribute your data across the shards in ways that supports the query patterns of your application.

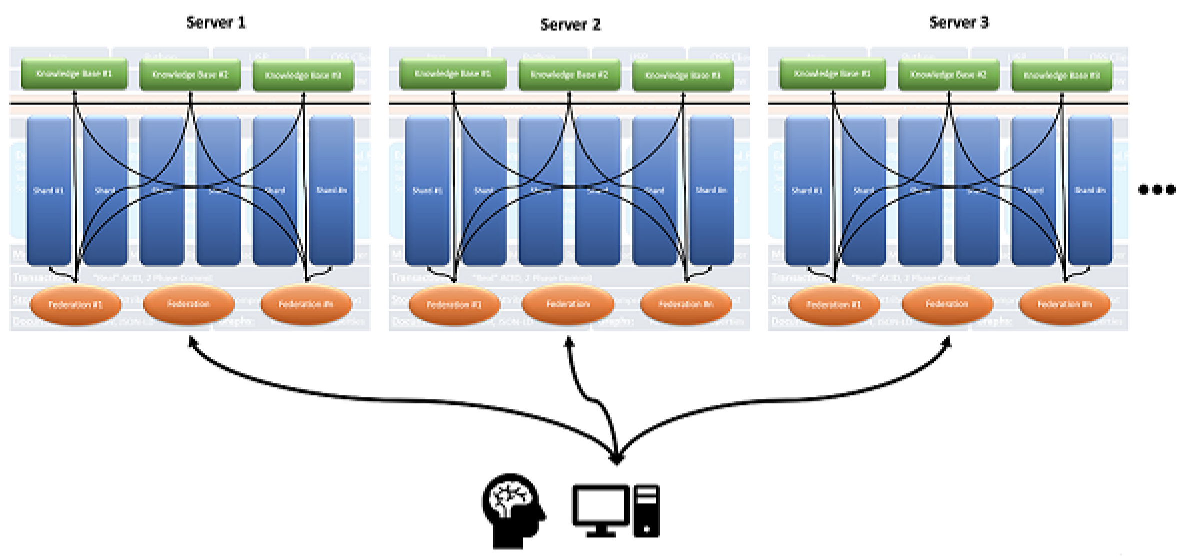
Data common to all shards is placed in knowledge base repositories which are federated with shards when queries are processed. This combination of shards and federated knowledge base repos, called FedShard™, accelerates results for highly complex queries.

This diagram shows how this works:



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

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



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

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

The basic setup

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

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

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

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

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

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

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

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

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

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

Suppose you have queries like:

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

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

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

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

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

Some data is common to all patients: the definition of conditions, doctors, and procedures. You may need to know these data when answering queries. Not if the query is How many patients were diagnosed with condition5678?' but if it is How many patients had a broken arm? as the latter requires knowing that condition5678` is a broken arm. Thus, triples like

```
condition5678 hasLabel "broken arm"
```

are needed on all shards so that queries like

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

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

Resource requirements

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

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

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

Daemonizing...

Server started with warning: When configured to serve a
```

distributed
database we suggest the soft file descriptor limit be 65536.
The
current limit is 1024.

Cluster Definition File

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

The distributed repository setup

A distributed repository has the following components:

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

repositories are created using specifications in the *agcluster.cfg* file. (To be clear about terminology: the *distributed repository definition* is the whole complex specified by the *agcluster.cfg* file: shards, kb repositories, and the distributed repository.)

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

these common kb repos.) You can have as many common repos as you like and need not have any.

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

The agcluster.cfg file

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

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

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

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

```
Toplevel directives
Collective directives
```

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

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

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

Three servers are defined:

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

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

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

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

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

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

```
Port 10035
Scheme http

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

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

The file defines:

- Three servers: servers are fully determined by a host (e.g. aghost1.franz.com), a port (10035, specified in a default directive line at the top), and a scheme (http or https, in this case http, specified in a default directive line at the top).
- A group of servers, specified in the group line with the label my-servers.
- A distributed repository named bigDB. This is specified

on the db line. A cluster repository with the name bigDB will be visible on each server after the distributed repository is defined and the databases are created.
- The key that will be used to determine which shard a triple is added to. key part graph says assign to a shard based on the graph of the triple. All triples with the same graph value end up in the same shard.
- A username and password. These should be valid for all servers in the my-servers group.
- An include directive saying the servers in the my-servers group should be used by the distributed repository.
- A shardsPerServer directive saying that each specified server will have three shards.

Comment lines

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

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

Labels

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

A label must precede references to it.

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

```
group my-servers
  server host1

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

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

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

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

Some more simple agcluster.cfg examples

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

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

```
Port 10035
Scheme http

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

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

We have specified two shard repositories, both on host1, one using the label host1 and one using the actual host name.

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

A note on constructed repository names

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

`<repository-name>.shard<index>`

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

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

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

The directives in the agcluster.cfg file

Defining directives

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

The server directive

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

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

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

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

```
server aghost.franz.com aghost
```

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

```
scheme https
port 12001
server aghost1.franz.com aghost1
```

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

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

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

Server-specific directives

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

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

password <password>
: The password for the user.

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

sudo <boolean>
: [Optional] Allow passwordless **sudo** on each host. **sudo** is necessary if, for example, you wish to install AllegroGraph into a directory that requires root privileges to write to.

bindir <directory>
: The directory where of the *bin/* subdirectory of the directory where AllegroGraph is installed on the server (the installation will be in the parent directory).

The repo directive

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

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

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

```
server aghost.franz.com aghost

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

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

Collective directives: GROUP and DB

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

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

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

The GROUP directive

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

The format is

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

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

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

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

repo-directives are formally defined above but in short are

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

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

server-directives are formally defined above but in short are

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

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

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

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

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

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

Here we include some repo directives

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

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

The DB directive

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

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

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

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

- **The db label**: this must be specified. It will name the distributed repository and might be used in naming shards not specifically named.
- **defaulting directives**: see the Defaulting directives section below. These directives can provide

default values for other directives. Any number of defaulting directives can be specified.

- **key**: this required directive specifies how triples should be assigned to shards. There are two arguments, the **type** and the **value**. The **type** is either *part* or *attribute*. The possible **values** for part are **subject**, **predicate**, **object**, and **graph**. The **value** for **attribute** is an attribute name. See the Triple Attributes document. For *key attribute name* all triples loaded into the distributed repository must have the the *name* attribute with a value. (All triples have a subject, predicate, object, and graph, the graph being the default graph is no graph is specified when the triple is added.) Only one key can be specified and once the distributed repository is created, it cannot be changed.
- **prefix**: string to be used when generating unique names for shards. It defaults to the **db** label. Only one prefix can be specified.
- **shardsPerServer**: a positive integer specifying the number of shards per server. The default is 1. Servers need not have the same number of shards and may have more shards than this value, but cannot have fewer. Thus all servers must have at least one shard. This directive can be specified once only.

- **kb**: a repo spec or label. This repository will be federated with each shard when processing a query. This repo typically contains ontology data and data which provides information elements of triples. Multiple kb directives can be specified.
- **include**: a group label. The servers in the group will be used in the distributed repository. Additional servers can be designated with the next directive. Multiple **include** directives can be specified.
- **server**: a server spec or label. This server will be

included in the distributed repository. Multiple **server** directives can be specified. Each **server** declaration will result in **shardsPerServer** shards being added, with names constructed from the **prefix** or the **db** label, without regard to **repo** declarations even if the repository spec supplied specifies the same server as a **server** directive. See the example below.
- **repo**: a repository spec or label. The repository will be included in the distributed repository. Multiple **repo** directives can be specified. **repo** directives add shards but they are not counted as the in shardsPerServer value, See the example below.

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

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

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

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

Here is an equivalent db directive specifying servers directly:

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

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

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

db my-cluster
  include my-servers
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

results in these shards:

will results in the following 3 shards:

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

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

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

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

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

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

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

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

of equivalently again:

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

Defaulting directives

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

- **port**: the port use by a server. Default when no **port** value is specified is 10035.
- **scheme**: the protocol to use when connecting to a server. The value must be http or https. Default when no **scheme** is specified in http.
- **catalog**: the catalog to use when resolving repos. Default when no **catalog** is specified is the root catalog.

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

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

```
Port 10035
Catalog my-catalog
Scheme https

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

The full server specification is

```
https://aghost1.franz.com:10035
```

The full repo specification is

```
https://aghost1.franz.com:10035/catalog/my-catalog/repositorie
s/my-repo
```

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

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

- **user** <username>
- **password** <password>
- **osuser** <name>
- **sudo** <boolean>
- **bindir** <directory>

Installing AllegroGraph on multiple servers

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

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

```
install-agraph --cluster-config agcluster.cfg --agraph-config
```

```
agraph.cfg [install-dir]
```

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

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

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

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

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

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

Changing the agcluster.cfg file

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

install-agraph –cluster-config [path of modified

```
copy/]agcluster.cfg
```

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

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

Starting and stopping the servers

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

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

All servers in a cluster can be started and stopped with

```
agraph-control --cluster start/stop
```

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

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

Using agtool utilities on a distributed repository

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

Using agtool export on a distributed repository

agtool export (see Repository Export) works on distributed repositories just like it does with regular repositories. All data in the various shards of the distributed repo is written to a regular data file which can be read into a regular repository or another distributed repo with the same number of shards or a different number of shards. (Nothing in the exported file indicates that the data came from a distributed repository).

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

Using agtool archive on a distributed repository

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

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

```
Port 19700
Scheme http

group my-servers
```

```
  server aghost1.franz.com host1
  server aghost2.franz.com host2

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

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

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

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

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

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

as follows:

```
% bin/agtool archive restore --newuuid
```

```
http://test:xyzzy@aghost1.franz.com:19700/repositories/restore
DB1 /aghost1/disk1/user1/drkenn1/ bigDB
agtool built with AllegroGraph Server
Restoring archive from /aghost1/disk1/user1/drkenn1/ to new
triple-store restorebigDB1
Restore throughput: 0.7 MB/s
Restore completed in 0h1m24s
Read 62.8 MiB from
/aghost1/disk1/user1/drkenn1/archives/root/bigDB/bigDB.agbacku
p
```

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

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

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

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

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

```
/aghost1/disk1/user1/drkenn1/archives/root/bigDB/bigDB.agbacku
p
%
```

Upgrading to a new version

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

Distributed repos in AGWebView

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

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

More information on running the cluster

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

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

Advanced Knowledge Graph Visualization with New Gruff v8

High Performance Data Visualizations Accelerate Graph Search and Query Building –   Driving Data Discoveries for Banks, Healthcare Providers and Enterprises Globally

OAKLAND, Calif., May 12, 2020 — Franz Inc., an early innovator in Artificial Intelligence (AI) and leading supplier of Semantic Graph Database technology for Knowledge Graph Solutions, today announced Gruff 8, a browser-based graph visualization software tool for exploring and discovering connections within enterprise Knowledge Graphs. Gruff 8, which has been integrated into AllegroGraph 7, enables users to visually build queries and visualize connections between data without writing code, which speeds discoveries and enhances the ability to uncover hidden connections within data.

“By augmenting Knowledge Graphs with visualizations, users can determine insights that would otherwise elude them,” said Jans Aasman, CEO of Franz Inc. “Gruff’s dynamic data visualizations increase users’ understanding of data by instantly illustrating relevant relationships, hidden patterns and data’s significance to outcomes. Gruff also helps make data actionable by displaying it in a way that decision-makers can see the significance of data relative to a business problem or solution.”

“Few tools exist that can quickly turn arbitrary RDF graph pattern matches into clear visualizable results,” said Michael Pool, Global Head of Semantic Modeling and Engineering, Senior Director at BNY Mellon Bank. “Gruff is invaluable in turning our knowledge graph data into useful and actionable analytic insights.”

Gruff enables users to create visual Knowledge Graphs that display data relationships in views that are driven by the user. Ad hoc and exploratory analysis can be performed by simply clicking on different graph nodes to answer questions. Gruff’s unique ‘Time Machine’ feature provides the capability to explore temporal context and connections within data. The visual query builder within Gruff empowers both novice and expert users to create simple to highly complex queries without writing any code.

Browser-based Graph Visualization – Gruff 8 is a browser-based application that does not require an additional download or application installation once AllegroGraph is installed. All AllegroGraph users need is a web browser and internet connection to login. This approach gives users the convenience to access Gruff from anywhere on any type of system, while also simplifying deployment and streamlining updates within enterprise environments.

Louis Rumanes at UnitedHealth Group Research and Development recognizes the value of using Gruff as a browser-based app and commented, “Nice job on Gruff in a browser and I think this will be a gamechanger.”

Accelerated Visual Graph Rendering – Visual renderings within Gruff are now up to 3X faster. Users can dynamically lay out cyclical graphs, display tables of properties and build SPARQL or Prolog queries as visual diagrams.

Dynamic Graph Visualizations within AllegroGraph – Gruff is fully integrated with AllegroGraph 7, Franz’s leading semantic knowledge graph solution, which seamlessly leverages Gruff’s advanced graph visualizations and graphical query builder to reveal hidden connections in knowledge graph data. AllegroGraph 7, with FedShard™, is a breakthrough Knowledge Graph solution that allows infinite data integration through a patented approach that unifies all data and knowledge base silos into an Entity-Event Knowledge Graph solution that can support massive big data analytics. AllegroGraph 7 utilizes unique federated sharding capabilities that drive 360-degree insights and enable complex reasoning across distributed Knowledge Graphs.

To support ubiquitous AI, a Knowledge Graph system needs to fuse and integrate data, not just in representation, but in context (ontologies, metadata, domain knowledge, terminology systems), and time (temporal relationships between components of data). The rich functional and contextual integration of multi-modal, predictive modeling, artificial intelligence suitable for large scale analytics is what distinguishes AllegroGraph 7 as a modern, scalable enterprise analytic platform.

AllegroGraph 7 is the first big temporal Knowledge Graph technology that encapsulates a novel entity-event model natively integrated with domain ontologies and metadata with dynamic ways of setting the analytics lens on all entities in the system (patient, person, devices, transactions, events, and operations) as prime objects that can be the focus of an analytic (AI, ML, DL) process.

“AllegroGraph 7’s support of Entity-Event Data Modeling is the most welcome innovation and addition to our arsenal in reimagining healthcare and implementing Precision Medicine,” said Dr. Parsa Mirhaji, Director of Center for Health Data Innovations at the Albert Einstein College of Medicine and Montefiore Health System, NY. “Precision Medicine is about moving away from statistical averages and broad-based patterns. It is about connecting many dots, from different contexts and throughout time, to support precision diagnosis and to recommend the precision care that can take into account all the subtle differences and nuisances of individuals and their personal experiences throughout their life. This technology is about saving lives, by leveraging data, context and analytics and is what Franz's Entity-Event Data Modeling brings to the table."

Gruff 8 Availability and Pricing

Guff 8 is immediately available as a free download from AllegroGraph.com and is integrated as part of AllegroGraph's cloud offering on the Amazon Marketplace.

Gruff Webinar
Join Franz's webcast discussing Gruff 8 entitled "Visualizing and Exploring Knowledge Graphs with the New Browser based Gruff" – by registering for the May 14th Webinar.

About Franz Inc.

Franz Inc. is an early innovator in Artificial Intelligence (AI) and leading supplier of Semantic Graph Database technology with expert knowledge in developing and deploying Knowledge Graph solutions. The foundation for Knowledge Graphs and AI lies in the facets of semantic technology provided by AllegroGraph and Allegro CL. AllegroGraph is a database technology that enables businesses to extract sophisticated decision insights and predictive analytics from highly complex, distributed data that cannot be uncovered with conventional databases. Unlike traditional relational databases or other NoSQL databases, AllegroGraph employs semantic graph technologies that process data with contextual and conceptual intelligence. AllegroGraph is able run queries of unprecedented complexity to support predictive analytics that help organizations make more informed, real-time decisions. AllegroGraph is utilized by dozens of the top F500 companies worldwide. To learn more about Franz and

AllegroGraph, go to www.franz.com.

Ubiquitous AI Demands A New Type Of Database Sharding

Forbes published the following article by Dr. Jans Aasman, Franz Inc.'s CEO.

The notion of sharding has become increasingly crucial for selecting and optimizing database architectures. In many cases, sharding is a means of horizontally distributing data; if properly implemented, it results in near-infinite scalability. This option enables database availability for business continuity, allowing organizations to replicate databases among geographic locations. It's equally useful for load balancing, in which computational necessities (like processing) shift between machines to improve IT resource allocation.

However, these use cases fail to actualize sharding's full potential to maximize database performance in today's post-big data landscape. There's an even more powerful form of sharding, called "hybrid sharding," that drastically improves the speed of query results and duly expands the complexity of the questions that can be asked and answered. Hybrid sharding is the ability to combine data that can be partitioned into shards with data that represents knowledge that is usually un-shardable.

This hybrid sharding works particularly well with the knowledge graph phenomenon leveraged by the world's top data-driven companies. Hybrid sharding also creates the enterprise scalability to query scores of internal and external sources for nuanced, detailed results, with responsiveness commensurate to that of the contemporary AI age.

Read the full article at Forbes.

NEW! – Franz's AllegroGraph 7 Powers First Distributed Semantic Knowledge Graph Solution with Federated-Sharding

FedShard™, Entity-Event Data Modeling and Browser-based Gruff Drives Infinite Data Integration, Holistic Insights and Complex Reasoning

Franz Inc., an early innovator in Artificial Intelligence (AI) and leading supplier of Semantic Graph Database technology for Knowledge Graph Solutions, today announced AllegroGraph 7, a breakthrough solution that allows infinite data integration through a patented approach unifying all data and siloed knowledge into an Entity-Event Knowledge Graph solution that can support massive big data analytics. AllegroGraph 7 utilizes unique federated sharding capabilities that drive 360-degree insights and enable complex reasoning across a distributed Knowledge Graph. Hidden connections in data are revealed to AllegroGraph 7 users through a new browser-based version of Gruff, an advanced visualization and graphical query builder.

“Large enterprises have Knowledge Graphs that are so big that no amount of vertical scaling will work,” said Jans Aasman, CEO of Franz Inc. “When these organizations want to conduct new big data analytics, it requires a new effort by the IT department to gather semi-usable data for the data scientists, which can cost millions of dollars, waste valuable time and still not provide a holistic data architecture for querying across all data. ETL, Data Lakes and Property Graphs only exacerbate the problem by creating new data silos. AllegroGraph 7 takes a holistic approach to mixed data, unifying all enterprise data with domain knowledge, including taxonomies, ontologies and industry knowledge – making queries across all data possible, while simplifying and accelerating feature extraction for machine learning.”

To support ubiquitous AI, a Knowledge Graph system will have to fuse and integrate data, not just in representation, but in context (ontologies, metadata, domain knowledge, terminology systems), and time (temporal relationships between components of data). The rich functional and contextual integration of multi-modal, predictive modeling and artificial intelligence is what distinguishes AllegroGraph 7 as a modern, scalable, enterprise analytic platform. AllegroGraph 7 is the first big temporal knowledge graph technology that encapsulates a novel entity-event model natively integrated with domain ontologies and metadata, and dynamic ways of setting the analytics lens on all entities in the system (patient, person, devices, transactions, events, and operations) as prime objects that can be the focus of an analytic (AI, ML, DL) process.

AI applications and complex reasoning analytics require information from both databases and knowledge bases that contain domain information, taxonomies and ontologies in order to conduct queries. Some large-scale knowledge bases cannot be sharded because they contain highly interconnected data. AllegroGraph 7 federates any shard with any large-scale knowledge base – providing a novel way to shard knowledge bases without duplicating knowledge bases in every shard. This approach creates a modern analytic system that integrates data in context (ontologies, metadata, domain knowledge, terminology systems) and time (temporal relationships between components of data). The result is a rich functional and contextual integration of data suitable for large scale analytics, predictive modeling, and artificial intelligence.

Financial institutions, healthcare providers, contact centers, manufacturing firms, government agencies and other large enterprises that use AllegroGraph 7 gain a holistic, future-proofed Knowledge Graph architecture for big data predictive analytics and machine learning across complex knowledge bases.

“AllegroGraph 7’s support of Entity-Event Data Modeling is the most welcome innovation and addition to our arsenal in reimagining healthcare and implementing Precision Medicine,” said Dr. Parsa Mirhaji, Director of Center for Health Data Innovations at the Albert Einstein College of Medicine and Montefiore Health System, NY “Precision Medicine is about moving away from statistical averages and broad-based patterns. It is about connecting many dots, from different contexts and throughout time, to support precision diagnosis and to recommend the precision care that can take into account all the subtle differences and nuisances of individuals and their personal experiences throughout their life. This technology is about saving lives, by leveraging data, context and analytics and is what Franz’s Entity-Event Data Modeling brings to the table.”

Dr. Mirhaji and his team at Montefiore Health System have developed the Patient-centered Analytic Learning Machine

(PALM) using these capabilities to provide an enterprise platform for Artificial Intelligence and machine learning in healthcare that can support conversational AI, interpret data from EMR, natural language, and radiological images, all centered around life-time experiences of an individual patient. A single system that unifies all analytics and data from heterogeneous sources to manage appointments and prescriptions, triage patients with potential spinal cancer, respiratory failure, or sepsis, and provide just-in-time recommendations and personalized decision support for clinicians to improve patients' outcomes.

Key capabilities in AllegroGraph 7 include:

Semantic Entity-Event Data Modeling

Big Data predictive analytics requires a new data model approach that unifies typical enterprise data with knowledge bases such as taxonomies, ontologies, industry terms and other domain knowledge. The Entity-Event Data Model utilized by AllegroGraph 7 puts core 'entities' such as customers, patients, students or people of interest at the center and then collects several layers of knowledge related to the entity as 'events.' The events represent activities that transpire in a temporal context. Using this novel data model approach, organizations gain a holistic view of customers, patients, students or important entities and the ability to discover deep connections, uncover new patterns and attain explainable results.

***FedShard*™ Speeds Complex Queries**

Through a patented in-memory federation function, the results from each machine are combined so that the query process appears as if only one database is being accessed, although many different databases and data stores and knowledge bases are actually being accessed and returning results. This unique data federation capability accelerates results for highly complex queries across highly distributed data sets and knowledge bases.

Large-scale Mixed Data Processing

The AllegroGraph 7 big data processing system is able to scale massive amounts of domain knowledge data by efficiently associating domain knowledge with partitioned data through shardable graphs on clusters of machines. AllegroGraph 7 efficiently combines partitioned data with domain knowledge through an innovative process that keeps as much of the data in RAM as possible to speed data access and fully utilize the processors of the query servers.

Browser-based Gruff
Gruff's powerful query and visualization capabilities are now available via a web browser and directly integrated in AllegroGraph 7. Gruff is the industry's leading Knowledge Graph visualization tool that dynamically displays visual graphs and related links. Gruff's 'Time Machine' provides users with an important capability to explore temporal connections and see how relationships are created over time. Users can build visual graphs that display the relationships in graph databases, display tables of properties, manage queries, connect to SPARQL Endpoints, and build SPARQL or Prolog queries as visual diagrams. Gruff can be downloaded separately or is included with the AllegroGraph v7 distribution.

High Performance Big Data Analytics

AllegroGraph 7 delivers high performance analytics by overcoming data processing issues related to disk versus memory access, uses processor core efficiency and updates domain knowledge databases across partitioned data systems in a highly efficient manner.

Gartner predicts "the application of graph processing and graph DBMSs will grow at 100 percent annually through 2022 to continuously accelerate data preparation and enable more complex and adaptive data science.” In addition, Gartner named graph analytics as a “Top 10 Data and Analytics Trend” to solve critical business priorities.” (*Source: Gartner, Top 10 Data and Analytics Trends, November 5, 2019)*

AllegroGraph 7 Availability

AllegroGraph 7 is immediately available directly from Franz Inc. Visit the AllegroGraph YouTube channel to see AllegroGraph in action.

Join AllegroGraph 7 Webinar
Franz Inc. will host a webcast entitled “Scalable Knowledge Graphs Using the New Distributed AllegroGraph 7.” Register for the Webinar.

Knowledge Graph Conference – May 4 – 7, 2020

Dr. Jans Aasman, CEO, Franz Inc., will be presenting a talk at the Knowledge Graph Conference entitled, “The Knowledge Graph that Listens” on May 7th at 1PM Eastern. Register for the Conference.

The Knowledge Graph Cookbook

Released April 22, 2020, this new book directs readers on why and how to build Knowledge Graphs that help enterprises use data to innovate, create value and increase revenue. The book is full of recipes and knowledge on the subject and features an interview with Dr. Jans Aasman, CEO, Franz Inc. in the Expert Opinion section. Get a copy of the book.

Natural Language Processing and Machine Learning in AllegroGraph

The majority of our customers build Knowledge Graphs with Natural Language and Machine learning components. Because of this trend AllegroGraph now offers strong support for the use of Natural Language Processing and Machine learning.

Franz Inc has a team of NLP engineers and Taxonomy experts that can help with building turn-key solutions. In general however, our customers already have some expertise in house. In those cases we train customers in how to take the output of NLP and ML processing and turn that into an efficient Knowledge Graph based on best practices in the industry.

This document primarily describes the NLP and ML plug-in AllegroGraph.

Note that many enterprises already have a data science team with NLP experts that use modern open source NLP tools like Spacy, Gensim or Polyglot, or Machine Learning based NLP tools like BERT and Scikit-Learn. In another blog about Document Handling we describe a pipeline of how to deal with NLP in Document Knowledge Graphs by using our NLP and ML plugin and mix that with open source tools.

PlugIn features for Natural Language Processing and Machine Learning in AllegroGraph.

Here is the outline of the plugin features that we are going to describe in more detail.

Machine learning

- data acquisition
- classifier training
- feature extraction support
- performance analysis
- model persistence

NLP

- handling languages
- handling dictionaries
- tokenization
- entity extraction
- Sentiment analysis
- basic pattern matching

SPARQL Access

- Future development

Machine Learning

ML: Data Acquisition
Given that the NLP and ML functions operate within AllegroGraph, after loading the plugins, data acquisition can be performed directly from the triple-store, which drastically simplifies the data scientist workflow. However, if the data is not in AllegroGraph yet we can also import it directly from ten formats of triples or we can use our additional capabilities to import from CSV/JSON/JSON-LD.

Part of the Data Acquisition is also that we need to pre-process the data for training so we provide these three functions:

- prepare-training-data
- split-dev-test

- equalize (for resampling)

Machine Learning: Classifiers

- Currently we provide simple linear classifiers. In case there’s a need for neural net or other advanced classifiers, those can be integrated on-demand.
- We also provide support for online learning (online machine learning is an ML method in which data becomes available in a sequential order and is used to update the best predictor for future data at each step, as opposed to batch learning techniques which generate the best predictor by learning on the entire training data set at once). This feature is useful for many real-world data sets that are constantly updated.
- The default classifiers available are Averaged Perceptron and AROW

Machine Learning: Feature Extraction

Each classifier is expecting a vector of features: either feature indices (indicative features) or pairs of numbers (index – value). These are obtained in a two-step process:

1. A classifier-specific extract-features method should be defined that will return raw feature vector with features identified by strings of the following form: prefix|feature.

The prefix should be provided as a keyword argument to the collect-features method call, and it is used to distinguish similar features from different sources (for instance, for distinct predicates).

2. Those features will be automatically transformed to unique integer ids. The resulting feature vector of indicator features may look like the following: #(1 123 2999 …)

Note that these features may be persisted to AllegroGraph for repeated re-use (e.g. for experimenting with classifier hyperparameter tuning or different classification models).

Many possible features may be extracted from data, but there is a set of common ones, such as:

1. individual tokens of the text field
2. ngrams (of a specified order) of the text field
3. presence of a token in a specific dictionary (like, the dictionary of slang words)
4. presence/value of a certain predicate for the subject of the current triple
5. length of the text

And in case the user has a need for special types of tokens we can write specific token methods, here is an example (in Lisp) that produces an indicator feature of a presence of emojis in the text:

```
(defmethod collect-features ((method (eql :emoji)) toks &key
pred)
(dolist (tok toks)
(when (some #'(lambda (code)
  (or (<= #x1F600 code #x1F64F)
      (<= #x1F650 code #x1F67F)
      (<= #x1F680 code #x1F6FF)))
   (map 'vector #'char-code tok))
(return (list "emoji")))))
```

Machine Learning: Integration with Spacy

The NLP and ML community invents new features and capabilities at an incredible speed. Way faster than any database company can keep up with. So why not embrace that? Whenever we need something that we don’t have in AllegroGraph yet we can call out to Spacy or any other external NLP tool. Here is an example of using feature extraction from Spacy to collect indicator features of the text dependency parse relations:

```
(defmethod collect-features ((method (eql :dep)) deps &key
pred dep-type dep-labels)
 (loop :for ds :in deps :nconc
  (loop :for dep :in ds
   :when (and (member (dep-tag dep) dep-labels)
              (dep-head dep)
              (dep-tok dep))
    :collect (format nil "dep|~a|~a_~a"
              dep-type
              (tok-word (dep-head dep)
              (tok-word (dep-tok dep))))))
```

The demonstrated integration uses Spacy Docker instance and its HTTP API.

Machine Learning: Classifier Analysis

We provide all the basic tools and metrics for classifier quality analysis:

- accuracy
- f1, precision, recall
- confusion matrix
- and an aggregated classification report

Machine Learning: Model Persistence

The idea behind model persistence is that all the data can be stored in AllegroGraph, including features and classifier models. AllegroGraph stores classifiers directly as triples. This is a far more robust and language-independent approach than currently popular among data scientists reliance on Python pickle files. For the storage we provide a basic triple-based format, so it is also possible to interchange the models using standard RDF data formats.

The biggest advantage of this approach is that when adding text to AllegroGraph we don’t have to move the data externally to perform the classification but can keep the whole pipeline entirely internal.

Natural Language Procession (NLP)

NLP: Language Packs

Most of the NLP tools are language-dependent: i.e. there’s a general function that uses language-specific model/rules/etc. In AllegroGraph, support for particular languages is provided on-demand and all the language-specific is grouped in the so called “language pack” or langpack, for short – a directory with a number of text and binary files with predefined names.

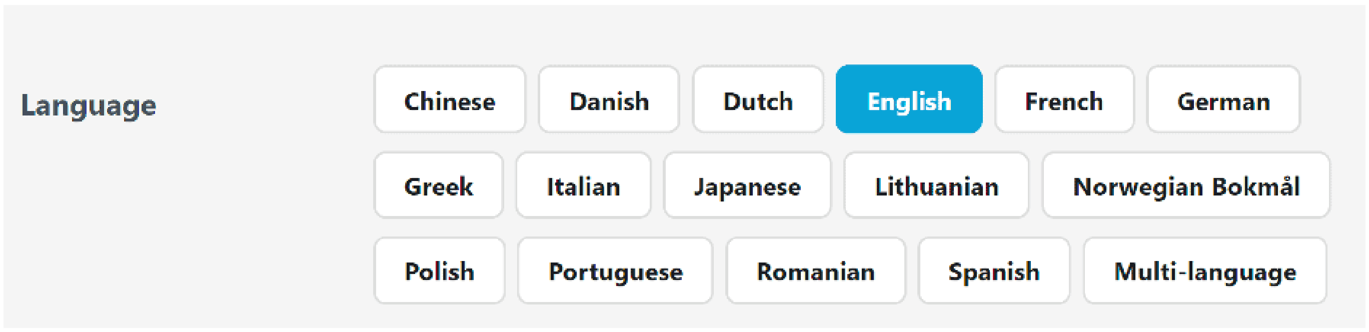
Currently, the langpack for English is provided at nlp/langs/en.zip, with the following files:

- contractions.txt – a dictionary of contractions
- abbrs.txt – a dictionary of abbreviations
- stopwords.txt – a dictionary of stopwords
- pos-dict.txt – positive sentiment words
- neg-dict.txt – negative sentiment words
- word-tok.txt – a list of word tokenization rules

Additionally, we use a general dictionary, a word-form dictionary (obtained from Wiktionary), and custom lexicons.

Loading a langpack for a particular language is performed using load-langpack.

Creating a langpack is just a matter of adding the properly named files to the directory and can be done manually. The names of the files should correspond to the names of the dictionary variables that will be filled by the pack. The dictionaries that don’t have a corresponding file will be just skipped.We have just finished creating a langpack for Spanish and it will be published soon. In case you need other dictionaries we use our AG/Spacy infrastructure. Spacy recently added a comprehensive list of new languages:



NLP: Dictionaries

Dictionaries are read from the language packs or other sources and are kept in memory as language-specific hash-tables. Alongside support for storing the dictionaries as text files, there are also utilities for working with them as triples and putting them into the triple store.

Note that we at Franz Inc specialize in Taxonomy Building using various commercial taxonomy building tools. All these tools can now export these taxonomies as a mix of SKOS taxonomies and OWL. We have several functions to read directly from these SKOS taxonomies and turn them into dictionaries that support efficient phrase-level lookup.

NLP: Tokenization

Tokenization is performed using a time-proven rule-based approach. There are 3 levels of tokenization that have both a corresponding specific utility function and an :output format of the tokenize function:

> :parags – splits the text into a list of lists of tokens for paragraphs and sentences in each paragraph
> :sents – splits the text into a list of tokens for each sentence
> :words – splits the text into a plain list of tokens

Paragraph-level tokenization considers newlines as paragraph delimiters. Sentence-level tokenization is geared towards western-style writing that uses dot and other punctuation marks to delimit sentences. It is, currently, hard-coded, but if the need arises, additional handling may be added for other writing systems. Word-level tokenization is performed using a language-specific set of rules.

NLP: Entity Extraction

Entity extraction is performed by efficient matching (exactly or fuzzy) of the token sequences to the existing dictionary structure.

It is expected that the entities come from the triple store and there’s a special utility function that builds lookup dictionaries from all the triples of the repository identified by certain graphs that have a skos:prefLabel or skos:altLabel property. The lookup may be case-insensitive with the exception of abbreviations (default) or case-sensitive.

Similar to entity extraction, there’s also support for spotting sentiment words. It is performed using the positive/negative words dictionaries from the langpack.

One feature that we needed to develop for our customers is ‘heuristic entity extraction’ . In case you want to extract complicated product names from text or call-center conversations between customers and agents you run into the problem that it becomes very expensive to develop altLabels in a taxonomy tool. We created special software to facilitate the automatic creation of altlabels.

NLP: Basic Pattern Matching for relationship and event detection

Getting entities out of text is now well understood and supported by the software community. However, to find complex concepts or relationships between entities or even events is way harder and requires a flexible rule-based pattern matcher. Given our long time background in Lisp and Prolog one can imagine we created a very powerful pattern matcher.

SPARQL Access

Currently all the features above can be controlled as stored procedures or using Lisp as the command language. We have a new (beta) version that uses SPARQL for most of the control. Here are some examples. Note that fai is a magic-property namespace for “AI”-related stuff and inc is a custom namespace of an imaginary client:

1. Entity extraction

```
select ?ent {
   ?subj fai:entityTaxonomy inc:products .
   ?subj fai:entityTaxonomy inc:salesTerms .
   ?subj fai:textPredicate inc:text .
   ?subj fai:entity(fai:language "en", fai:taxonomy inc:products) ?ent .
}
```

The expressions ?subj fai:entityTaxonomy inc:poducts and ?subj fai:entityTaxonomy inc:salesTerms specify which taxonomies to use (the appropriate matchers are cached).
The expression ?subj fai:entity ?ent will either return the already extracted entities with the specified predicate (fai:entity) or extract the new entities according to the taxonomies in the texts accessible by fai:textPredicate.

2. fai:sentiment will return a single triple with sentiment score:

```
select ?sentiment {
   ?subj fai:textPredicate inc:text .
   ?subj fai:sentiment ?sentiment .
   ?subj fai:language "en" .
   ?subj fai:sentimentTaxonomy franz:sentiwords .
}
```

3. Text classification:
Provided inc:customClassifier was already trained previously, this query will return labels for all texts as a result of classification.

```
select ?label {
?subj fai:textPredicate inc:text .
?subj fai:classifier inc:customClassifier .
?subj fai:classify ?label .
?label fai:storeResultPredicate inc:label .
}
```

Further Development
Our team is currently working on these new features:

- A more accessible UI (python client & web) to facilitate NLP and ML pipelines
- Addition of various classifier models
- Sequence classification support (already implemented for a customer project)
- Pre-trained models shipped with AllegroGraph (e.g. English NER)
- Graph ML algorithms (deepwalk, Google Expander)
- Clustering algorithms (k-means, OPTICS)

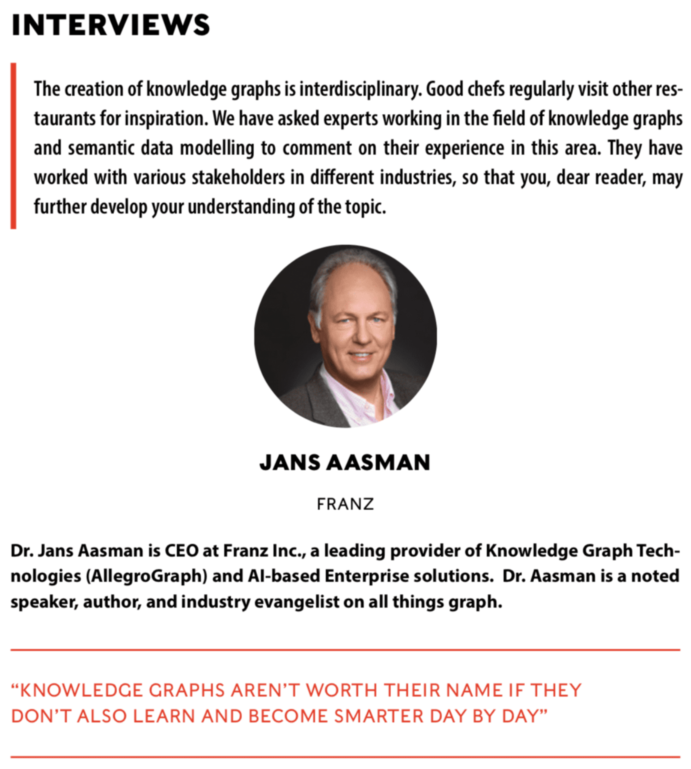
The Knowledge Graph Cookbook

Recipes for Knowledge Graphs that Work:

- Learn why and how to build knowledge graphs that help enterprises use data to innovate, create value and increase revenue. This practical manual is full of recipes and knowledge on the subject.
- Learn more about the variety of applications based on knowledge graphs.
- Learn how to build working knowledge graphs and which technologies to use.
- See how knowledge graphs can benefit different parts of your organization.
- Get ready for the next generation of enterprise data management tools.

Dr. Jans Aasman, CEO, Franz Inc. is interviewed in the Expert Opinion Section.

> **“KNOWLEDGE GRAPHS AREN’T WORTH THEIR NAME IF THEY DON’T ALSO LEARN AND BECOME SMARTER DAY BY DAY”** – Dr. Aasman


INTERVIEWS

The creation of knowledge graphs is interdisciplinary. Good chefs regularly visit other restaurants for inspiration. We have asked experts working in the field of knowledge graphs and semantic data modelling to comment on their experience in this area. They have worked with various stakeholders in different industries, so that you, dear reader, may further develop your understanding of the topic.

JANS AASMAN

FRANZ

Dr. Jans Aasman is CEO at Franz Inc., a leading provider of Knowledge Graph Technologies (AllegroGraph) and AI-based Enterprise solutions. Dr. Aasman is a noted speaker, author, and industry evangelist on all things graph.

"KNOWLEDGE GRAPHS AREN'T WORTH THEIR NAME IF THEY DON'T ALSO LEARN AND BECOME SMARTER DAY BY DAY"

Click here to get the book as free PDF or Kindle version.

Answering the Question Why: Explainable AI

The statistical branch of Artificial Intelligence has enamored organizations across industries, spurred an immense amount of capital dedicated to its technologies, and entranced numerous media outlets for the past couple of years. All of this attention, however, will ultimately prove unwarranted unless organizations, data scientists, and various vendors can answer one simple question: can they provide Explainable AI?

Although the ability to explain the results of Machine Learning models—and produce consistent results from them—has never been easy, a number of emergent techniques have recently appeared to open the proverbial ‘black box’ rendering these models so difficult to explain.

One of the most useful involves modeling real-world events with the adaptive schema of knowledge graphs and, via Machine Learning, gleaning whether they’re related and how frequently they take place together.

When the knowledge graph environment becomes endowed with an additional temporal dimension that organizations can traverse forwards and backwards with dynamic visualizations, they can understand what actually triggered these events, how one affected others, and the critical aspect of causation necessary for Explainable AI.

Read the full article at AIthority.

---

100 Companies That Matter in Knowledge Management

Franz Inc., is proud to announce that it has been named to The 100 Companies That Matter in Knowledge Management by KMWorld. The annual list reflects the urgency felt among many organizations to provide a timely flow of targeted information. Among the more prominent initiatives is the use of AI and cognitive computing, as well as related capabilities such as machine learning, natural language processing, and text analytics.

"Knowledge management software and services providers are embracing a fresh wave of technological innovation to address heightened expectations—among both customers and employees—for the right information to be delivered to the right people at the right time, said Tom Hogan, Group Publisher at KMWorld. "To showcase organizations that are advancing their products and capabilities to meet changing requirements, KMWorld created the annual list of 100 Companies That Matter in Knowledge Management."

"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."

How To Avoid Another AI Winter

Forbes published the following article by Dr. Jans Aasman, Franz Inc.'s CEO.

Photo: GETTY

Although there has been great progress in artificial intelligence (AI) over the past few years, many of us remember the AI winter in the 1990s, which resulted from overinflated promises by developers and unnaturally high expectations from end users. Now, industry insiders, such as Facebook head of AI Jerome Pesenti, are predicting that AI will soon hit another wall—this time due to the lack of semantic understanding.

“Deep learning and current AI, if you are really honest, has a lot of limitations,” said Pesenti. “We are very, very far from human intelligence, and there are some criticisms that are valid: It can propagate human biases, it’s not easy to explain, it doesn’t have common sense, it’s more on the level of pattern matching than robust semantic understanding.”

Read the full article at Forbes.