Comex AI Conference 2021 (Recording)

COMEX is the largest Technology, Communications, Innovation and Digital Transformation show in the Sultanate of Oman and offers exhibitors and visitors a comprehensive and highly specialized platform for industry leading discussions, knowledge-sharing, B2B meetings.

Dr. Sheng-Chuan Wu presented — Using AI for Practical Business Applications

---

Adding Properties to Triples in AllegroGraph

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

1.0 The Named Graph for Properties

Semantic Graph Databases are actually defined by the W3C standard to store RDF as 'Quads' (Named Graph, Subject,

Predicate, and Object). The 'Triple Store' terminology has stuck even though the industry has moved on to storing quads. We believe using the named graph approach to store metadata about triples is richer model that the property graph database method.

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

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

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

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

Here is the data displayed in Gruff, AllegroGraph's associated triple store browser:
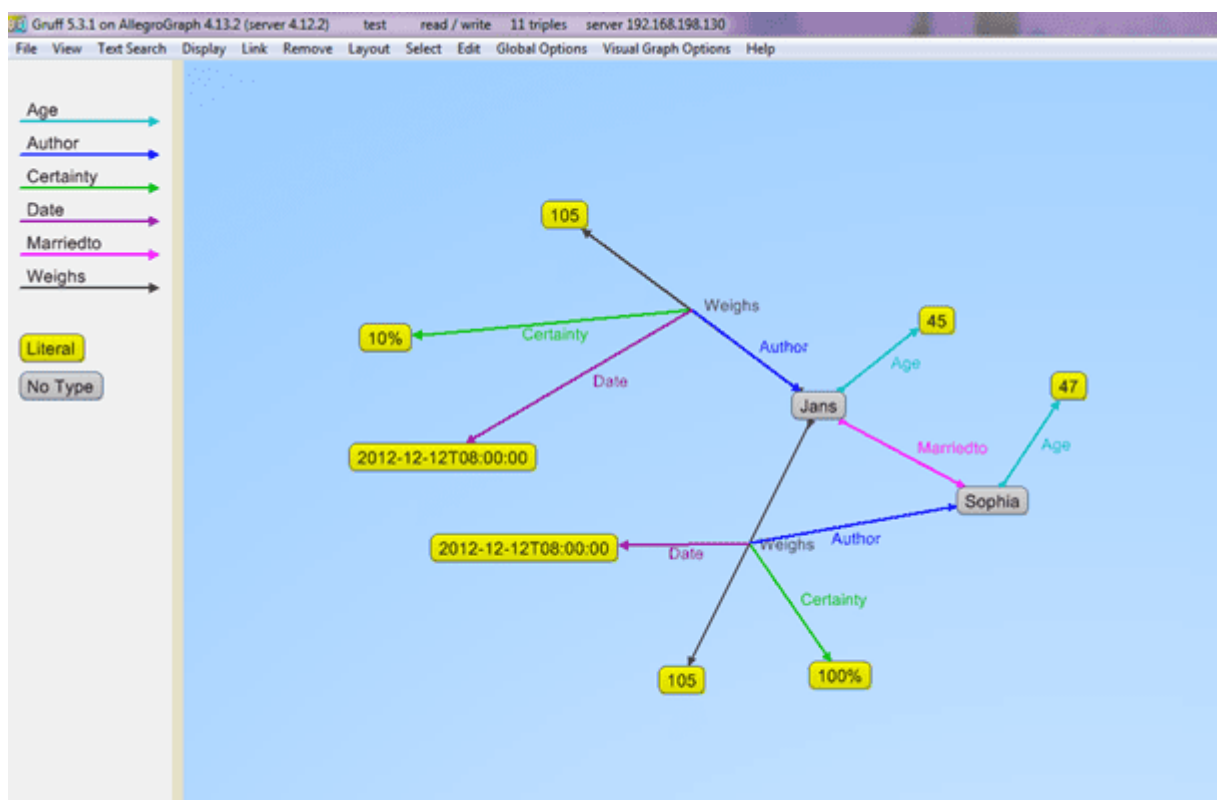
Using named graphs for a triple's metadata is a powerful tool but it does have limitations: (1) only one graph value can be associated with a triple, (2) it can be important that metadata is stored directly and physically with the triple (with named graphs, the actual metadata is usually stored in additional triples with the graph as the subject, as in the example above), and (3) named graphs have competing uses and

may not be available for metadata.

2.0 The Triple Attributes approach

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

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

Example with Attributes:

Subject — <http://dbpedia.org/resource/Arif_Babayev>
Predicate — <http://dbpedia.org/property/placeOfDeath>
Object — <http://dbpedia.org/resource/Baku>
Named Graph — <http://ex#trans@@1142684573200001>
    Triple Attributes — {"securityLevel": "high", "department": "hr", "accessToken": ["E", "D"]}

This article provides an initial introduction to attributes and the associated concept of static filters, showing how they are set up and used. We start with a security example which also describes the basics of adding attributes to triples and

filtering query results based on attribute values. Then we discuss other potential uses of attributes.

2.1 Triple Attribute Basics: a Security Example

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

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

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

AllegroGraph WebView

repository repo

| Repository | Queries | Utilities | Admin | User test

Repository repo — 0 statements

[edit description]

Load and Delete Data
- Add a statement
- Delete statements
- Import RDF:
    - from an uploaded file
    - from a server-side file
    - from a text area input

Explore the Repository
- View triples
- View quads
- View repository's classes
- View repository's predicates
- View repository's named graphs

Reports
- Storage report
- Triple indices
- String table
- Full list of reports ...

Multi-Master Replication
- Convert store to a replication instance

Warm Standby Replication
- Control replication

Repository Control
- Export repository as [N-Triples ▼]
- Start a session — support transactions and Prolog functors
- Warmup store
- Back-up this repository
- Export duplicate statements [Subject, Predicate, Object and Graph (spog) ▼]
- Delete duplicate statements
- Suppress duplicate statements false
- View active transactions
- Recognize geospatial datatypes automatically: ☐
- Control durability (bulk-load mode)
- Manage attribute definitions
- Set static attribute filter
- Manage triple indices
    - Optimize the repository ❓

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

AllegroGraph WebView   repository repo

| Repository | Queries | Utilities | Admin | User test

New Attribute

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

Save   Cancel

Attribute Definitions

+ Add New

Name	Min. number	Max. number	Allowed values	Ordered	

No attributes have been defined

We click Save and the attribute is defined:

AllegroGraph WebView   repository repo

| Repository | Queries | Utilities | Admin | User test

Attribute Definitions

+ Add New

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

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

Static Filter

Current filter

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

Edit filter

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

Save   Revert to current   Clear

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

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

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

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

                                   <http://www.franz.com#emp1>
<http://www.franz.com#position> "worker" {"security-level":
"2"} .

                                   <http://www.franz.com#emp2>
<http://www.franz.com#position> "manager" {"security-level":
"3"} .

                                   <http://www.franz.com#emp3>
<http://www.franz.com#position> "boss" {"security-level": "4"}
.
```

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

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

AllegroGraph WebView   repository repo

↺ | Repository | Queries | Utilities | Admin | User test

Edit query

```
1 # View triples
2 SELECT ?s ?p ?o { ?s ?p ?o . }
```

[Execute] [Log Query] [Show Plan]   [Save] as [_____]   [Add to repository]

No results

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

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

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

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

so the query should be

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

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

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

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

Edit query

```
1  prefix franzOption_userAttributes: <franz:%7B%22security-level%22%3A%223%22%7D>
2  select ?s ?p ?o { ?s ?p ?o . }
3
```

Execute | Log Query | Show Plan | Save as [   ] | Add to repository

| 1 Result in 2.619 ms | Information |

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

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

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

we get three results:

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

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

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

Let us redefine the filter to:

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

AllegroGraph WebView   repository repo

↰ | **Repository** | **Queries** | **Utilities** | **Admin** | **User test**

Static Filter

Current filter

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

Edit filter

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

Save | Revert to current | Clear

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

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

The triple

  emp0 position   "intern"

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

"user".

It would be a useful feature that we could associate attributes with actual users. However, this is not as simple as it sounds. Attributes are features of repositories. If I have a REPO1 repository, it can have a bunch of defined attributes and filters but my REPO2 may know nothing about them and its triples may not have any attributes at all, and no attributes are defined, and (as a result) no filters. But users are not repository-linked objects. While a repository can be made read-only or unreadable for a user, users do not have finer repository features. So an interface for providing users with attributes, since it would only make sense on a per-repository basis, requires a complicated interface. That is not yet implemented (though we are considering how it can be done).

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

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

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

2.2 Using attributes as additional data

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

Attribute Definitions

+ Add New

Name	Min. number	Max. number	Allowed values	Ordered	
timeAdded	1	1			⊗

Now let us define some triples:

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

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

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

AllegroGraph WebView   repository at-test

↶ | **Repository | Queries | Utilities | Admin | User test**

Repository at-test — 5 statements

[edit description]

Load and Delete Data

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

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

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

Edit query

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

Execute | Log Query | Show Plan | Save as [   ] | Add to repository

5 Results in 0.363 ms | Information

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

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

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

Edit query

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

Execute   Log Query   Show Plan   Save as [ ]   Add to repository

4 Results in 0.630 ms	Information

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

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

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

emp0 hasSalary 50000

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

emp0 hasSalary 55000

But that is not satisfactory because we have lost the salary

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

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


emp0 hasSalaryChange salaryChange017
```

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

```
<http://www.franz.com#emp0>  <http://www.franz.com#hasSalary>
"50000"^^<http://www.w3.org/2001/XMLSchema#integer>
{"timeAdded": "2017-01-12T10:12:45" } .
<http://www.franz.com#emp0>  <http://www.franz.com#hasSalary>
"55000"^^<http://www.w3.org/2001/XMLSchema#integer>
{"timeAdded": "2019-03-17T12:00:00" } .
```

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

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

Edit query

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

Execute | Log Query | Show Plan | Save as [   ] | Add to repository

1 Result in 0.388 ms | Information

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

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

```
select ?o ?value {
                      (?ta      ?value)
<http://franz.com/ns/allegrograph/6.2.0/attributesNameValue>

                      (<http://www.franz.com#emp0>
<http://www.franz.com#hasSalary> ?o) .
    }  order by desc(?value)
```

Edit query

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

Execute Log Query Show Plan Save as [_____] Add to repository

2 Results in 0.413 ms Information

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

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

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

Unraveling the Quandary of Access Layer versus Storage Layer Security

Dr. Jans Aasman was quoted in this article about how AllegroGraph's Triple Attributes provide Storage Layer Security.

With horizontal standards such as the General Data Protection Regulation (GDPR) and vertical mandates like the Fair Credit Reporting Act increasing in scope and number, information security is impacted by regulatory compliance more than ever.

Organizations frequently decide between concentrating protection at the access layer via role-based security filtering, or at the storage layer with methods like encryption, masking, and tokenization.

The argument is that the former underpins data governance policy and regulatory compliance by restricting data access according to department or organizational role. However, the latter's perceived as providing more granular security implemented at the data layer.

*A hybrid of access based security and security at the data layer—implemented by triple attributes—can counteract the weakness of each approach with the other's strength, resulting in information security that **Franz** CEO Jans Aasman characterized as "fine-grained and flexible enough" for any regulatory requirements or security model.*

*The security provided by this semantic technology is considerably enhanced by the addition of **key-value pairs** as JSON objects, which can be arbitrarily assigned to triples within databases. These key-value pairs provide a second security mechanism "embedded in the storage, so you cannot cheat," Aasman remarked.*

When implementing HIPPA standards with triple attributes, "even if you're a doctor, you can only see a patient record if all your other attributes are okay," Aasman mentioned.

"We're talking about a very flexible mechanism where we can add any combination of key-value pairs to any triples, and have a very flexible language to specify how to use that to create flexible security models," Aasman said.

Read the full article at InfoSecurity.

Solving Knowledge Graph Data Prep with Standards

Dataversity – December 2018

There's a general consensus throughout the data ecosystem that Data Preparation is the most substantial barrier to capitalizing on data-driven processes. Whether organizations are embarking on Data Science initiatives or simply feeding

any assortment of enterprise applications, the cleansing, classifying, mapping, modeling, transforming, and integrating of data is the most time honored (and time consuming) aspect of this process.

Approximately 80 percent of the work of data scientists is mired in Data Preparation, leaving roughly 20 percent of their jobs to actually exploiting data. Moreover, the contemporary focus on external sources, Big Data, social and mobile technologies has exploded the presence of semi-structured and unstructured data, which accounts for nearly 80 percent of today's data and further slows the preparation processes.

Read the full article at Dataversity.

The Most Secure Graph Database Available

Triples offer a way of describing model elements and relationships between them. In come cases, however, it is also convenient to be able to store data that is associated with a triple as a whole rather than with a particular element. For instance one might wish to record the source from which a triple has been imported or access level necessary to include it in query results. Traditional solutions of this problem include using graphs, RDF reification or triple IDs. All of these approaches suffer from various flexibility and performance issues. For this reason AllegroGraph offers an alternative: triple attributes.

Attributes are key-value pairs associated with a triple. Keys refer to attribute definitions that must be added to the store before they are used. Values are strings. The set of legal values

of an attribute can be constrained by the definition of that attribute. It is possible to associate multiple values of a given attribute with a single triple.
Possible uses for triple attributes include:

- *Access control: It is possible to instruct AllegroGraph to prevent an user from accessing triples with certain attributes.*
- *Sharding: Attributes can be used to ensure that related triples are always placed in the same shard when AllegroGraph acts as a distributed triple store.*

Like all other triple components, attribute values are immutable. They must be provided when the triple is added to the store and cannot be changed or removed later.
To illustrate the use of triple attributes we will construct an artificial data set containing a log of information about contacts detected by a submarine at a single moment in time.

Managing attribute definitions

Before we can add triples with attributes to the store we must create appropriate attribute definitions.
First let's open a connection

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

Attribute definitions are represented by **AttributeDefinition** objects. Each definition has a name, which must be unique, and a few optional properties (that can also be passed as constructor arguments):

- *allowed_values: a list of strings. If this property is set then only the values from this list can be used for the defined attribute.*
- *ordered: a boolean. If true then attribute value comparisons will use the ordering defined by allowed_values. The default is false.*
- *minimum_number, maximum_number: integers that can be used to constrain the cardinality of an attribute. By default there are no limits.*

Let's define a few attributes that we will later use to demonstrate various attribute-related capabilities of AllegroGraph. To do this, we will use the **setAttributeDefinition()** method of the connection object.

```python
from franz.openrdf.repository.attributes import AttributeDefinition

# A simple attribute with no constraints governing the set
# of legal values or the number of values that can be
# associated with a triple.
tag = AttributeDefinition(name='tag')

# An attribute with a limited set of legal values.
# Every bit of data can come from multiple sources.
# We encode this information in triple attributes,
# since it refers to the tripe as a whole. Another
# way of achieving this would be to use triple ids
# or RDF reification.
source = AttributeDefinition(
    name='source',
    allowed_values=['sonar', 'radar', 'esm', 'visual'])

# Security level - notice that the values are ordered
# and each triple *must* have exactly one value for
# this attribute. We will use this to prevent some
# users from accessing classified data.
level = AttributeDefinition(
    name='level',
    allowed_values=['low', 'medium', 'high'],
    ordered=True,
    minimum_number=1,
    maximum_number=1)

# An attribute like this could be used for sharding.
# That would ensure that data related to a particular
# contact is never partitioned across multiple shards.
# Note that this attribute is required, since without
# it an attribute-sharded triple store would not know
# what to do with a triple.
contact = AttributeDefinition(
    name='contact',
    minimum_number=1,
    maximum_number=1)

# So far we have created definition objects, but we
# have not yet sent those definitions to the server.
# Let's do this now.
conn.setAttributeDefinition(tag)
conn.setAttributeDefinition(source)
```

```
conn.setAttributeDefinition(level)
conn.setAttributeDefinition(contact)

# This line is not strictly necessary, because our
# connection operates in autocommit mode.
# However, it is important to note that attribute
# definitions have to be committed before they can
# be used by other sessions.
conn.commit()
```

It is possible to retrieve the list of attribute definitions from a repository by using the **getAttributeDefinitions()** method:

```python
for attr in conn.getAttributeDefinitions():
    print('Name: {0}'.format(attr.name))
    if attr.allowed_values:
        print('Allowed values: {0}'.format(
            ', '.join(attr.allowed_values)))
        print('Ordered: {0}'.format(
            'Y' if attr.ordered else 'N'))
    print('Min count: {0}'.format(attr.minimum_number))
    print('Max count: {0}'.format(attr.maximum_number))
    print()
```

Notice that in cases where the maximum cardinality has not been explicitly defined, the server replaced it with a default value. In practice this value is high enough to be interpreted as 'no limit'.

```
Name: tag
Min count: 0
Max count: 1152921504606846975

Name: source
Allowed values: sonar, radar, esm, visual
Min count: 0
Max count: 1152921504606846975
Ordered: N

Name: level
Allowed values: low, medium, high
Ordered: Y
Min count: 1
Max count: 1

Name: contact
Min count: 1
Max count: 1
```

Attribute definitions can be removed (provided that the attribute is not used by the static attribute filter, which will be discussed later) by calling **deleteAttributeDefinition()**:

```
conn.deleteAttributeDefinition('tag')
defs = conn.getAttributeDefinitions()
print(', '.join(sorted(a.name for a in defs)))
```

```
contact, level, source
```

Adding triples with attributes

Now that the attribute definitions have been established we can demonstrate the process of adding triples with attributes. This can be achieved using various methods. A common element of all these methods is the way in which triple attributes are represented. In all cases dictionaries with attribute names as keys and strings or lists of strings as values are used.

When **addTriple()** is used it is possible to pass attributes in a keyword parameter, as shown below:

```
ex = conn.namespace('ex://')
conn.addTriple(ex.S1, ex.cls, ex.Udaloy, attributes={
    'source': 'sonar',
    'level': 'low',
    'contact': 'S1'
})
```

The **addStatement()** method works in similar way. Note that it is not possible to include attributes in the **Statement** object itself.

```
from franz.openrdf.model import Statement

s = Statement(ex.M1, ex.cls, ex.Zumwalt)
conn.addStatement(s, attributes={
    'source': ['sonar', 'esm'],
    'level': 'medium',
    'contact': 'M1'
})
```

When adding multiple triples with **addTriples()** one can add a fifth element to each tuple to represent attributes. Let us illustrate

this by adding an aircraft to our dataset.

```
conn.addTriples(
    [(ex.R1, ex.cls, ex['Ka-27'], None,
      {'source': 'radar',
       'level': 'low',
       'contact': 'R1'}),
     (ex.R1, ex.altitude, 200, None,
      {'source': 'radar',
       'level': 'medium',
       'contact': 'R1'})])
```

When all or most of the added triples share the same attribute set it might be convenient to use the `attributes` keyword parameter. This provides default values, but is completely ignored for all tuples that already contain attributes (the dictionaries are not merged). In the example below we add a triple representing an aircraft carrier and a few more triples that specify its position. Notice that the first triple has a lower security level and multiple sources. The common 'contact' attribute could be used to ensure that all this data will remain on a single shard.

```
conn.addTriples(
    [(ex.M2, ex.cls, ex.Kuznetsov, None, {
        'source': ['sonar', 'radar', 'visual'],
        'contact': 'M2',
        'level': 'low',
     }),
     (ex.M2, ex.position, ex.pos343),
     (ex.pos343, ex.x, 430.0),
     (ex.pos343, ex.y, 240.0)],
    attributes={
        'contact': 'M2',
        'source': 'radar',
        'level': 'medium'
    })
```

Another method of adding triples with attributes is to use the NQX file format. This works both with **addFile()** and **addData()** (illustrated below):

```
from franz.openrdf.rio.rdfformat import RDFFormat

conn.addData('''
    <ex://S2> <ex://cls> <ex://Alpha> \
    {"source": "sonar", "level": "medium", "contact": "S2"} .
    <ex://S2> <ex://depth> "300" \
```

```
        {"source": "sonar", "level": "medium", "contact": "S2"} .
        <ex://S2> <ex://speed_kn> "15.0" \
        {"source": "sonar", "level": "medium", "contact": "S2"} .
    ''', rdf_format=RDFFormat.NQX)
```

When importing from a format that does not support attributes, it is possible to provide a common set of attribute values with a keyword parameter:

```python
from franz.openrdf.rio.rdfformat import RDFFormat

conn.addData('''
    <ex://V1> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 100 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V2> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 200 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V3> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 300;
              <ex://speed_kn> 12.0e+8 .
    <ex://V4> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 400 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V5> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 500 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V6> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 600 ;
              <ex://speed_kn> 12.0e+8 .
''', attributes={
    'source': 'visual',
    'level': 'high',
    'contact': 'a therapist'})
```

The data above represents six visually observed Walrus-class submarines, flying at different altitudes and well above the speed of light. It has been highly classified to conceal the fact that someone has clearly been drinking while on duty — after all there are only four Walrus-class submarines currently in service, so the observation is obviously incorrect.

Retrieving attribute values

We will now print all the data we have added to the store, including attributes, to verify that everything worked as

expected. The only way to do that is through a SPARQL query using the appropriate magic property to access the attributes. The query below binds a literal containing a JSON representation of triple attributes to the *?a* variable:

```python
import json

r = conn.executeTupleQuery('''
    PREFIX attr: <http://franz.com/ns/allegrograph/6.2.0/>
    SELECT ?s ?p ?o ?a {
        ?s ?p ?o .
        ?a attr:attributes (?s ?p ?o) .
    } ORDER BY ?s ?p ?o''')
with r:
    for row in r:
        print(row['s'], row['p'], row['o'])
        print(json.dumps(json.loads(row['a'].label),
                         sort_keys=True,
                         indent=4))
```

The result contains all the expected triples with pretty-printed attributes.

```
<ex://M1> <ex://cls> <ex://Zumwalt>
{
    "contact": "M1",
    "level": "medium",
    "source": [
        "esm",
        "sonar"
    ]
}
<ex://M2> <ex://cls> <ex://Kuznetsov>
{
    "contact": "M2",
    "level": "low",
    "source": [
        "visual",
        "radar",
        "sonar"
    ]
}
<ex://M2> <ex://position> <ex://pos343>
{
    "contact": "M2",
    "level": "medium",
    "source": "radar"
}
```

```
<ex://R1> <ex://altitude> "200"^^...
{
    "contact": "R1",
    "level": "medium",
    "source": "radar"
}
<ex://R1> <ex://cls> <ex://Ka-27>
{
    "contact": "R1",
    "level": "low",
    "source": "radar"
}
<ex://S1> <ex://cls> <ex://Udaloy>
{
    "contact": "S1",
    "level": "low",
    "source": "sonar"
}
<ex://S2> <ex://cls> <ex://Alpha>
{
    "contact": "S2",
    "level": "medium",
    "source": "sonar"
}
<ex://S2> <ex://depth> "300"
{
    "contact": "S2",
    "level": "medium",
    "source": "sonar"
}
<ex://S2> <ex://speed_kn> "15.0"
{
    "contact": "S2",
    "level": "medium",
    "source": "sonar"
}
<ex://V1> <ex://altitude> "100"^^...
{
    "contact": "a therapist",
    "level": "high",
    "source": "visual"
}
<ex://V1> <ex://cls> <ex://Walrus>
{
    "contact": "a therapist",
    "level": "high",
    "source": "visual"
}
<ex://V1> <ex://speed_kn> "1.2E9"^^...
{
```

```
        "contact": "a therapist",
        "level": "high",
        "source": "visual"
    }
    ...
    <ex://pos343> <ex://x> "4.3E2"^^...
    {
        "contact": "M2",
        "level": "medium",
        "source": "radar"
    }
    <ex://pos343> <ex://y> "2.4E2"^^...
    {
        "contact": "M2",
        "level": "medium",
        "source": "radar"
    }
```

Attribute filters

Triple attributes can be used to provide fine-grained access control. This can be achieved by using static attribute filters. Static attribute filters are simple expressions that control which triples are visible to a query based on triple attributes. Each repository has a single, global attribute filter that can be modified using **setAttributeFilter()**. The values passed to this method must be either strings (the syntax is described in the documentation of static attribute filters) or filter objects.
Filter objects are created by applying set operators to 'attribute sets'. These can then be combined using filter operators.
An attribute set can be one of the following:

- *a string or a list of strings: represents a constant set of values.*
- *TripleAttribute.name: represents the value of the name attribute associated with the currently inspected triple.*
- *UserAttribute.name: represents the value of the name attribute associated with current query. User attributes will be discussed in more detail later.*

Available set operators are shown in the table below. All classes and functions mentioned here can be imported from the `franz.openrdf.repository.attributes` package:

Syntax	Meaning
`Empty(x)`	True if the specified attribute set is empty.
`Overlap(x, y)`	True if there is at least one matching value between the two attribute sets.
`Subset(x, y)`, `x << y`	True if every element of *x* can be found in *y*
`Superset(x, y)`, `x >> y`	True if every element of *y* can be found in *x*
`Equal(x, y)`, `x == y`	True if *x* and *y* have exactly the same contents.
`Lt(x, y)`, `x < y`	True if both sets are singletons, at least one of the sets refers to a triple or user attribute, the attribute is ordered and the value of the single element of *x* occurs before the single value of *y* in the `lowed_values` list of the attribute.
`Le(x, y)`, `x <= y`	True if *y* < *x* is false.
`Eq(x, y)`	True if both *x* < *y* and *y* < *x* are false. Note that using the == Python operator translates to *Eqauls*, not *Eq*.
`Ge(x, y)`, `x >= y`	True if *x* < *y* is false.
`Gt(x, y)`, `x > y`	True if *y* < *x*.

Note that the overloaded operators only work if at least one of the attribute sets is a `UserAttribute` or `TripleAttribute` reference — if both arguments are strings or lists of strings the default Python semantics for each operator are used. The prefix syntax always produces filters.

Filters can be combined using the following operators:

Syntax	Meaning
Not(x), ~x	Negates the meaning of the filter.
And(x, y, ...), x & y	True if all subfilters are true.
Or(x, y, ...), x | y	True if at least one subfilter is true.

Filter operators also work with raw strings, but overloaded operators will only be recognized if at least one argument is a filter object.

Using filters and user attributes

The example below displays all classes of vessels from the dataset after establishing a static attribute filter which ensures that only sonar contacts are visible:

```python
from franz.openrdf.repository.attributes import *

conn.setAttributeFilter(TripleAttribute.source >> 'sonar')
conn.executeTupleQuery(
    'select ?class { ?s <ex://cls> ?class } order by ?class',
    output=True)
```

The output contains neither the visually observed Walruses nor the radar detected ASW helicopter.

```
-----------------
| class         |
=================
| ex://Alpha    |
| ex://Kuznetsov |
| ex://Udaloy   |
| ex://Zumwalt  |
-----------------
```

To avoid having to set a static filter before each query (which would be inefficient and cause concurrency issues) we can employ user attributes. User attributes are specific to a particular connection and are sent to the server with each query. The static attribute filter can refer to these and compare them with triple attributes. Thus we can use code presented below to create a filter which ensures that a connection only accesses data at or

below the chosen clearance level.

```
conn.setUserAttributes({'level': 'low'})
conn.setAttributeFilter(
    TripleAttribute.level <= UserAttribute.level)
conn.executeTupleQuery(
    'select ?class { ?s <ex://cls> ?class } order by ?class',
    output=True)
```

We can see that the output here contains only contacts with the access level of *low*. It omits the destroyer and Alpha submarine (these require *medium* level) as well as the top-secret Walruses.

```
------------------
| class          |
==================
| ex://Ka-27     |
| ex://Kuznetsov |
| ex://Udaloy    |
------------------
```

The main advantage of the code presented above is that the filter can be set globally during the application setup and access control can then be achieved by varying user attributes on connection objects.
Let us now remove the attribute filter to prevent it from interfering with other examples. We will use the **clearAttributeFilter()** method.

```
conn.clearAttributeFilter()
```

It might be useful to change connection's attributes temporarily for the duration of a single code block and restore prior attributes after that. This can be achieved using the **temporaryUserAttributes()** method, which returns a context manager. The example below illustrates its use. It also shows how to use **getUserAttributes()** to inspect user attributes.

```
with conn.temporaryUserAttributes({'level': 'high'}):
    print('User attributes inside the block:')
    for k, v in conn.getUserAttributes().items():
        print('{0}: {1}'.format(k, v))
    print()
print('User attributes outside the block:')
for k, v in conn.getUserAttributes().items():
    print('{0}: {1}'.format(k, v))
```

```
User attributes inside the block:
level: high

User attributes outside the block:
level: low »
```