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.

```
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',
```

```python
        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()**:

```python
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:

```python
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:

```
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:

```
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.

Syntax	Meaning
`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.

```python
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.

```python
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 »
```