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

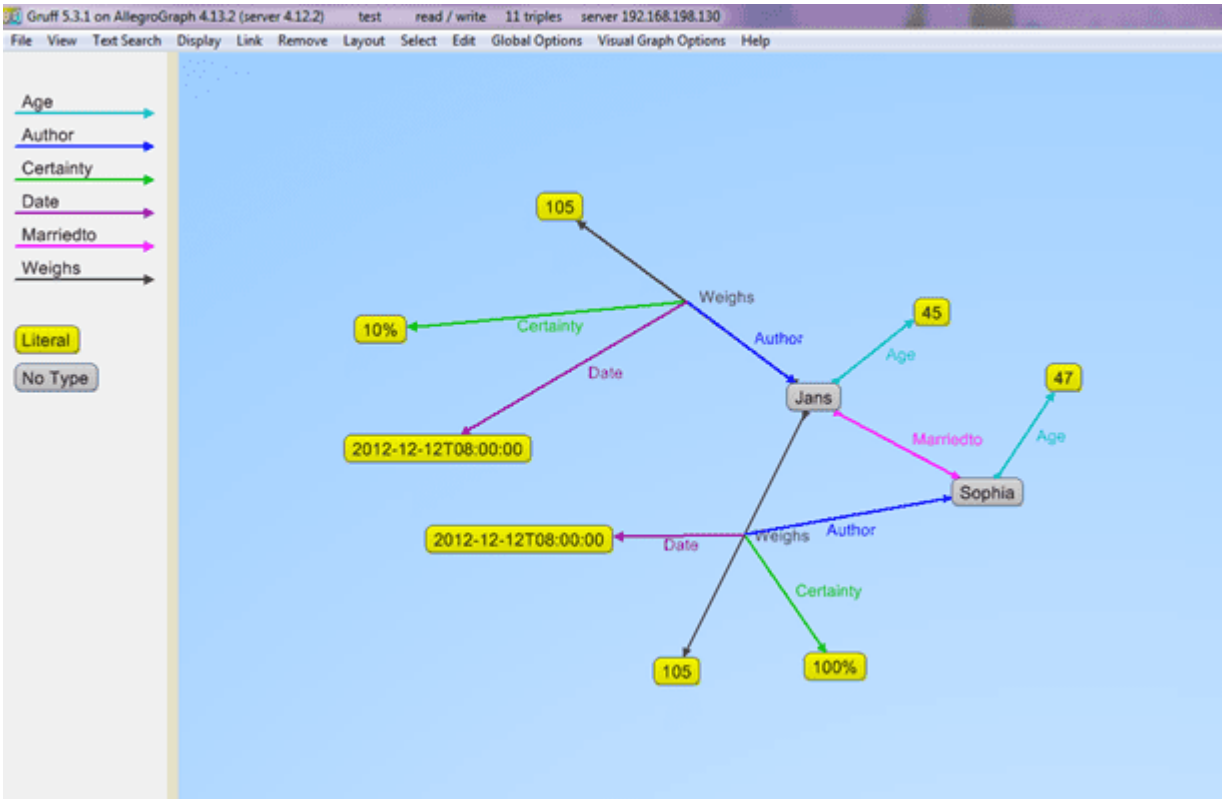
- **The author of the first triple (with graph S1) is Sophia and the author of the second (with graph S2) is Bruce (who is also the subject of the two triples).**

- **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.

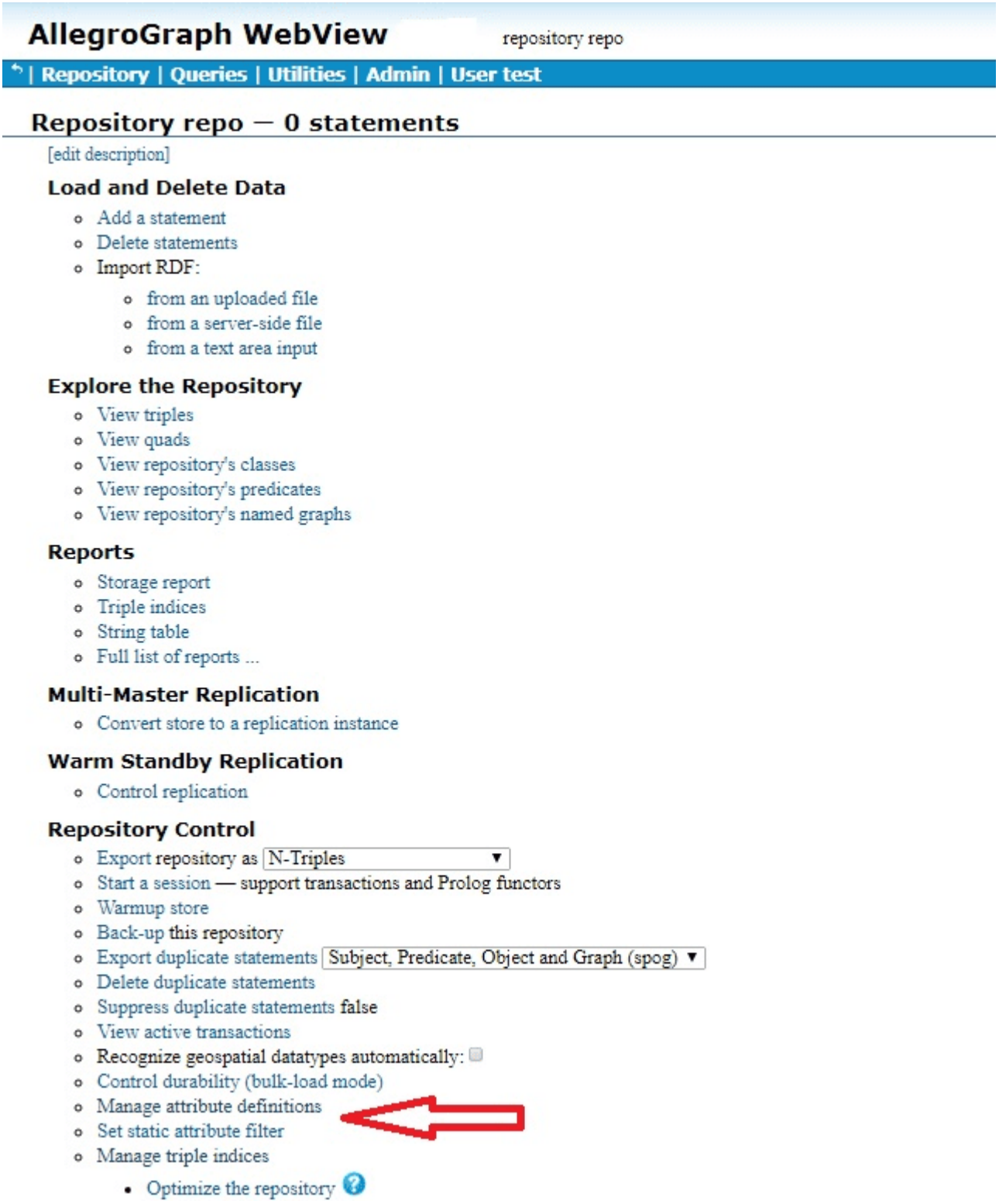
2.1 Triple Attribute Basics: a Security Example

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.

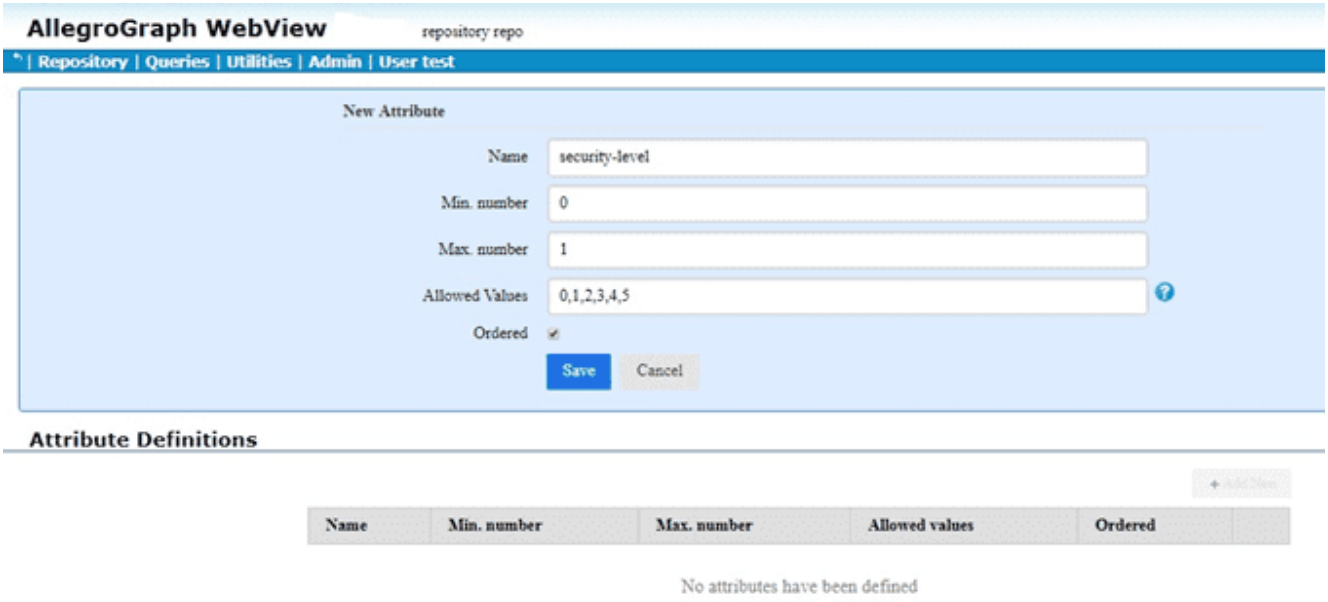
Note that users as such do not have attributes. Instead, attribute values are assigned when a query is posed. This is

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.

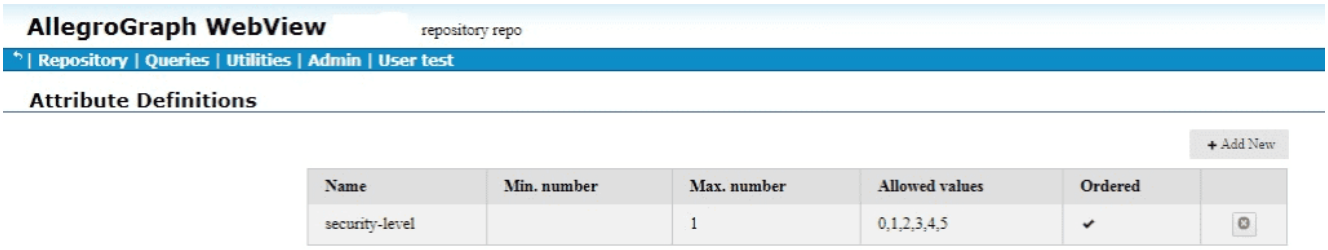
Here is a simple example. We define attributes and static attribute filters using AGWebView. We have a repository named repo. Here is a portion of its AGWebView page:



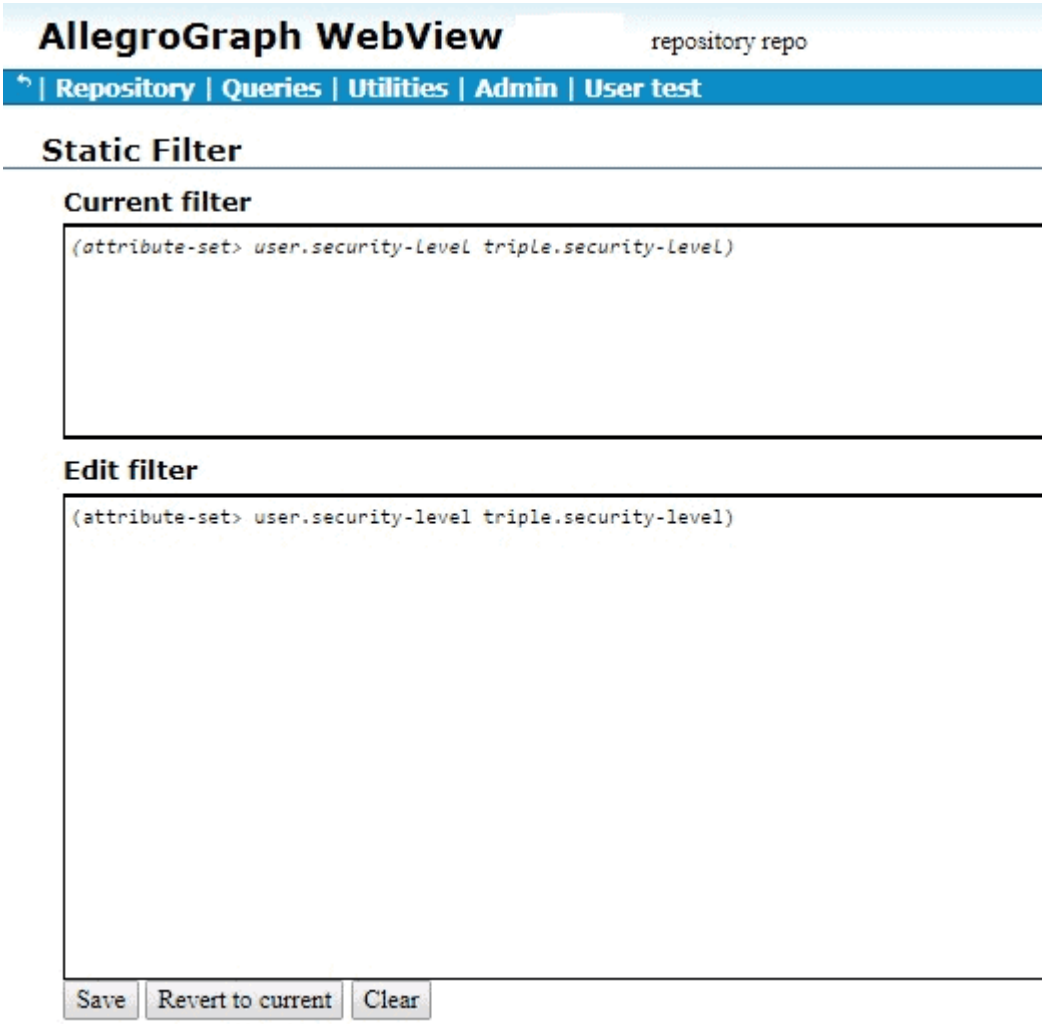
The red arrow points to the commands of interest: Manage attribute definitions and Set static attribute filter. We click on Set static attribute filter to define an attribute. We have filled in the attribute information (name *security-level*, minimum and maximum number allowed per triple, allowed values,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



We click Save and the attribute is defined:



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



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” .

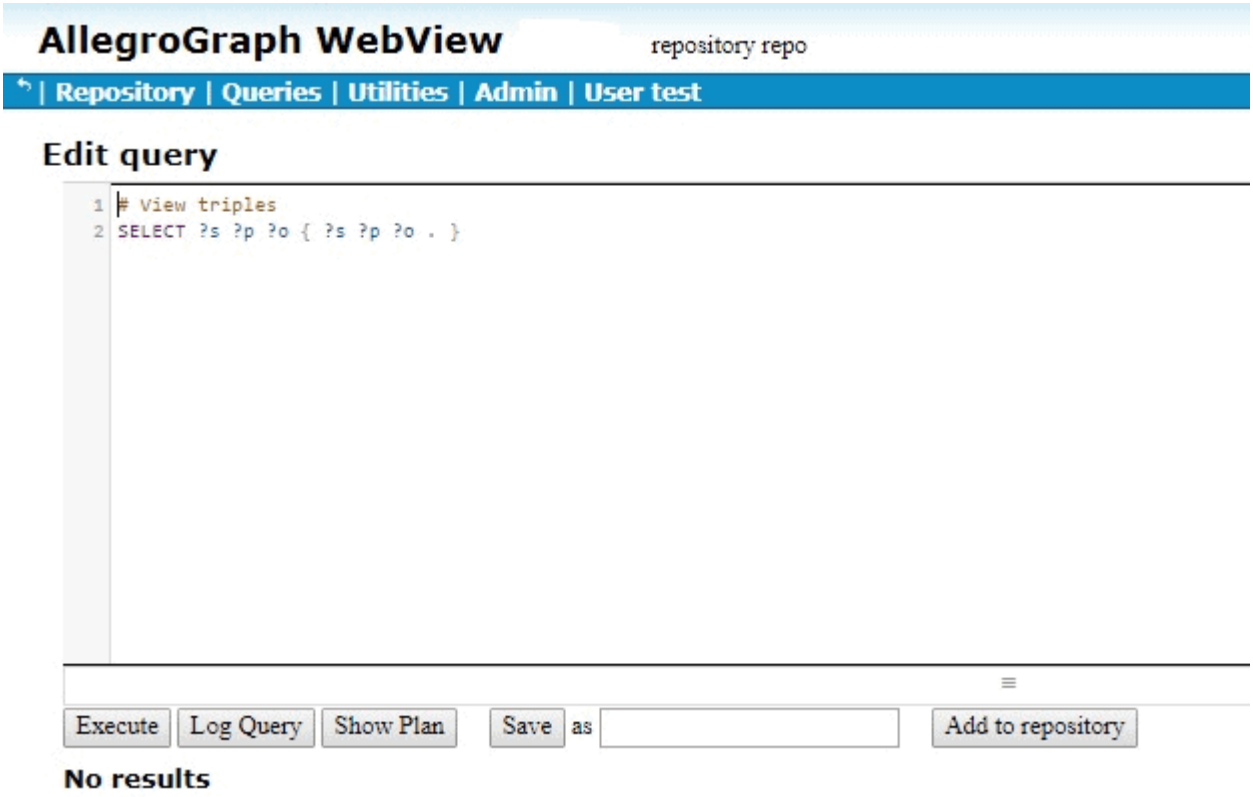
<http://www.franz.com#emp1> <http://www.franz.com#position> “worker” {“security-level”: “2”} .

<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:



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-
```

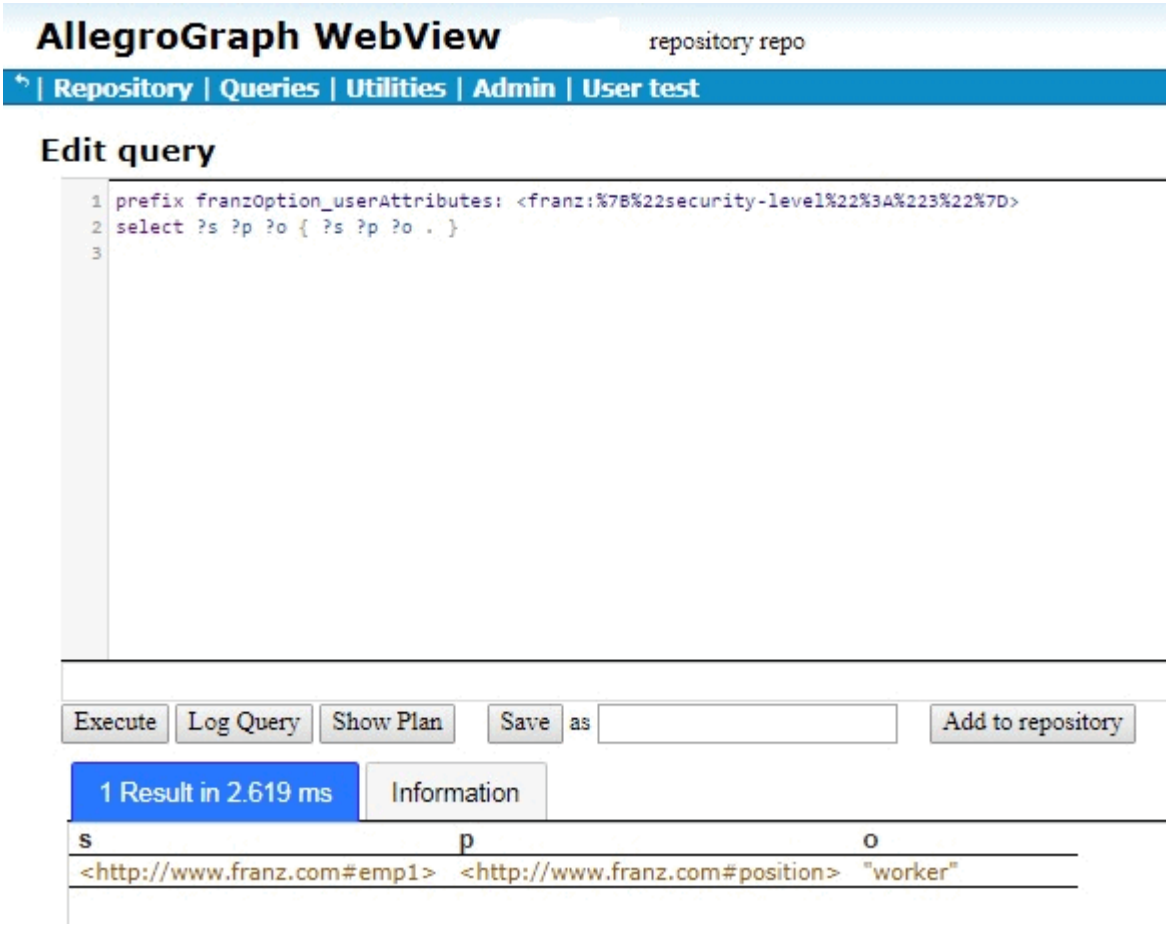
level%22%3A%223%22%7D>

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:



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:

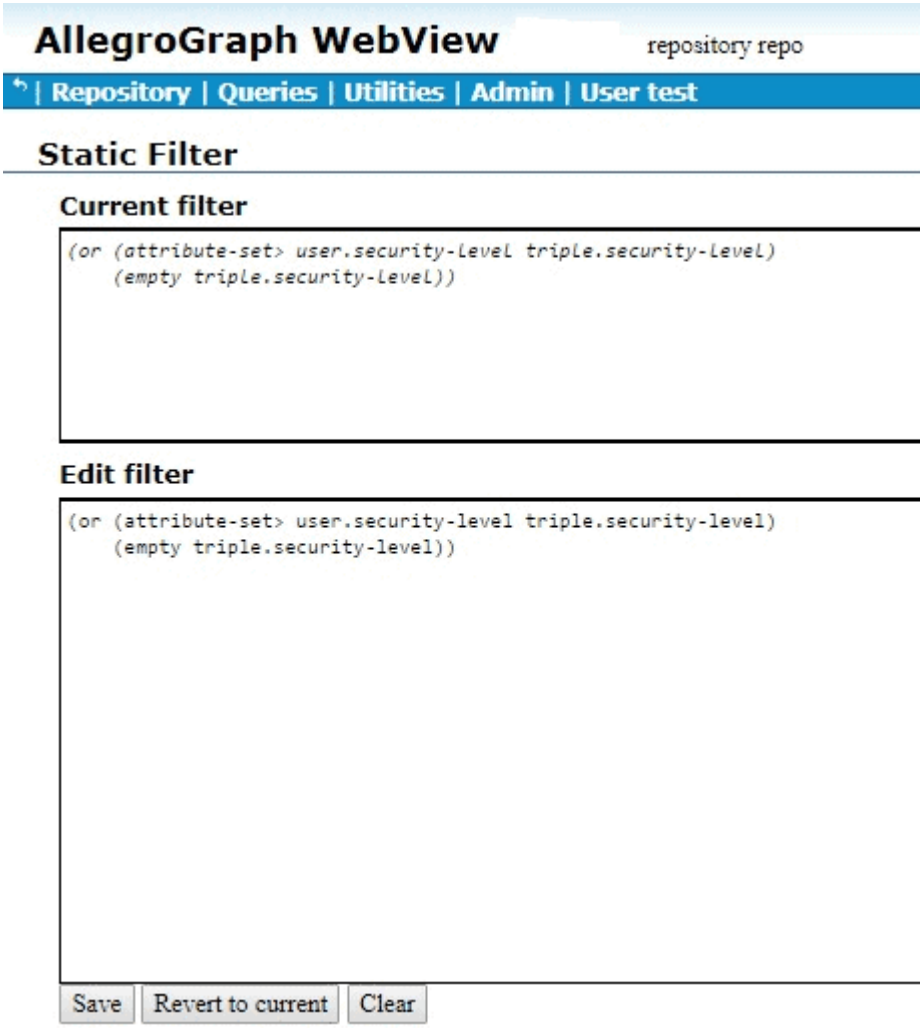
emp3	**position**	**“boss”**
emp2	**position**	**“manager”**
emp1	**position**	**“worker”**

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.

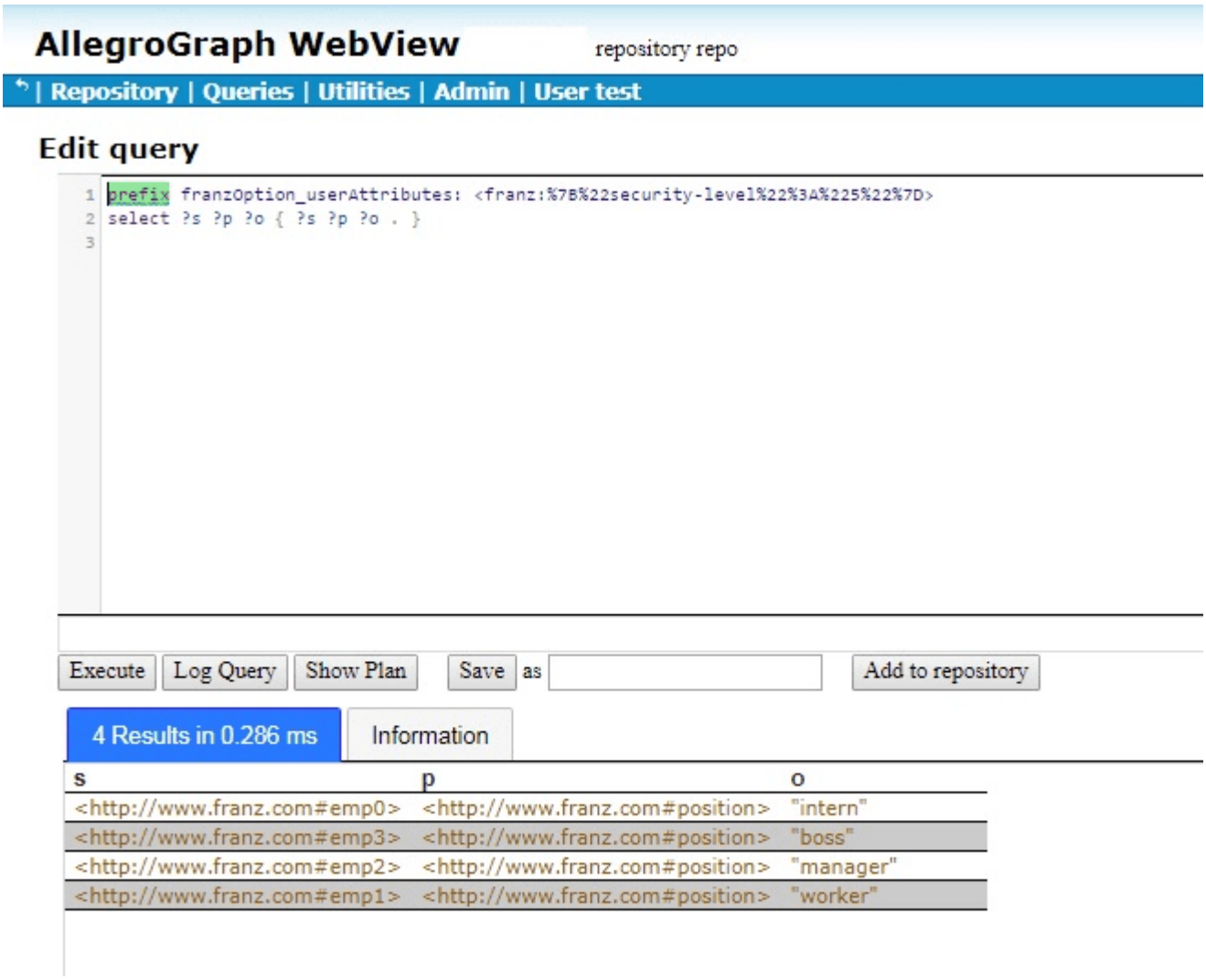
Let us redefine the filter to:

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



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).

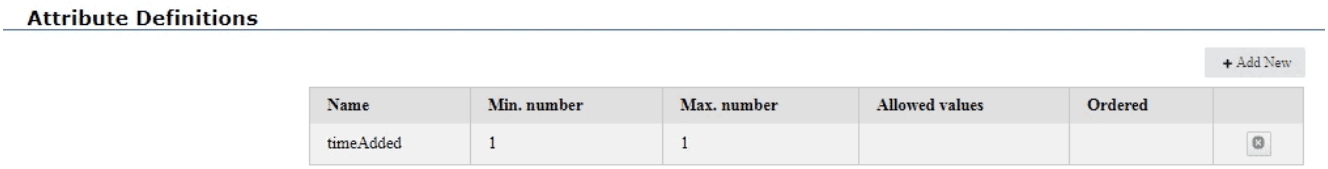
Instead, users can have specific prefixes associated with them and that prefix and be included in any query made by the user.

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:



Now let us define some triples:

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

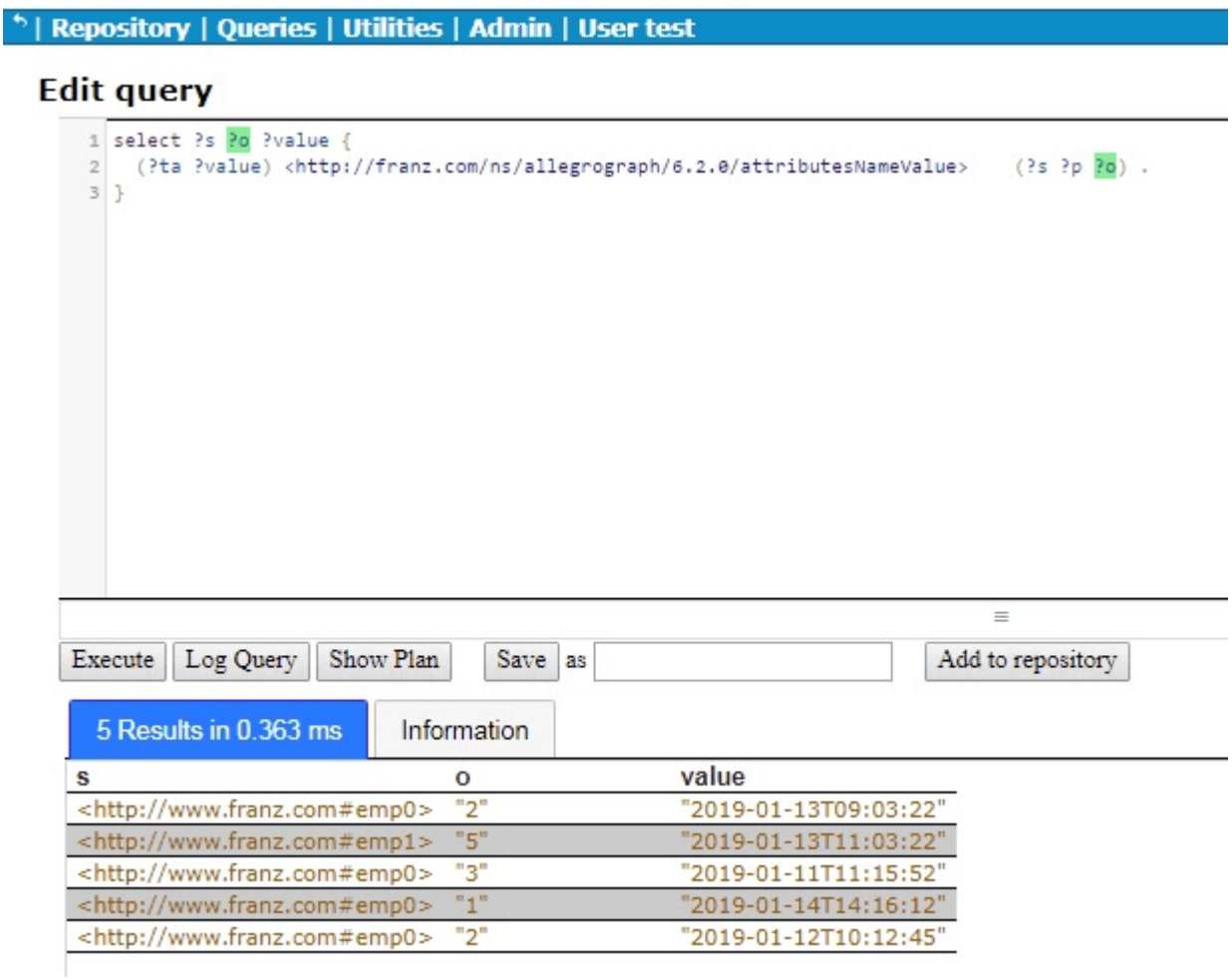
```
     <http://www.franz.com#emp1>
<http://www.franz.com#callRank> “5” {“timeAdded”:
“2019-01-13T11:03:22” } .
     <http://www.franz.com#emp0>
<http://www.franz.com#callRank> “2” {“timeAdded”:
“2019-01-13T09:03:22” } .
```

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

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

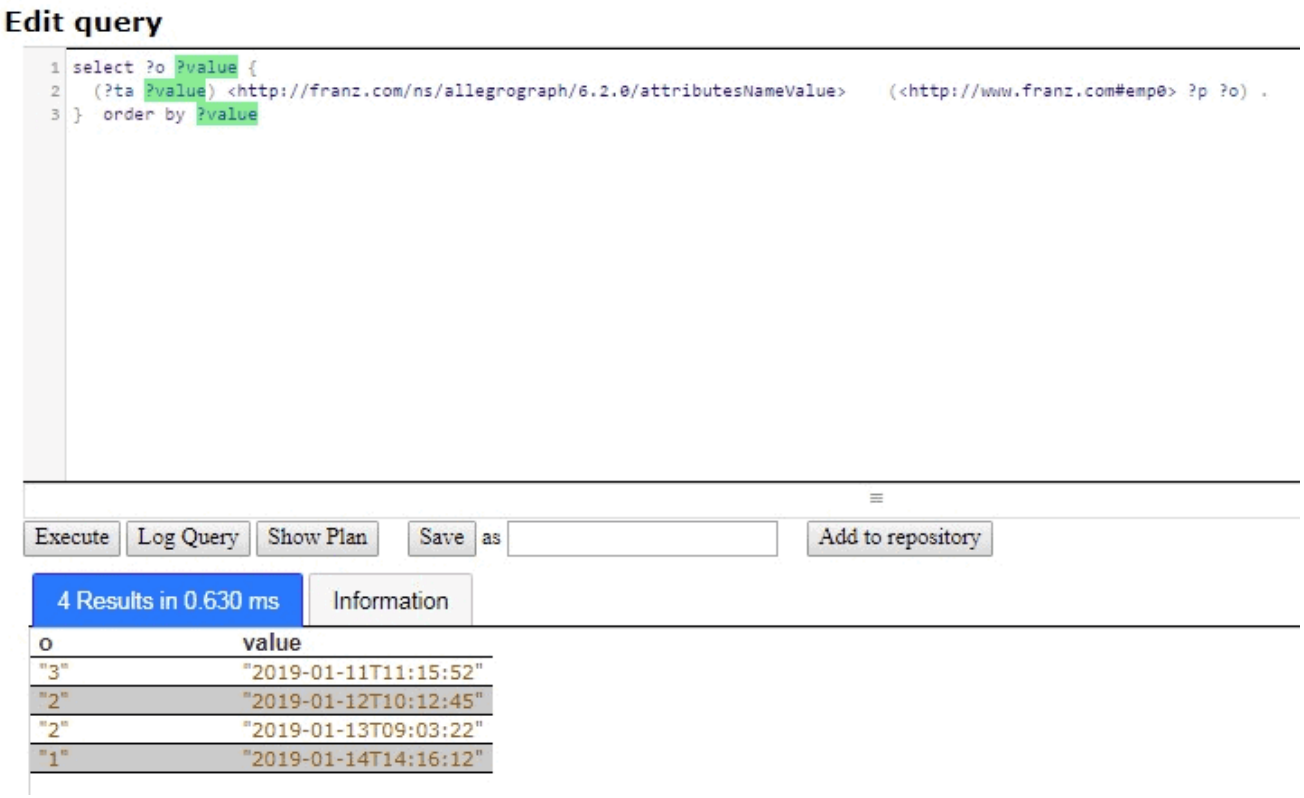
```
     select ?s ?o ?value {
                                        (?ta      ?value)
```

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



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
```



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:

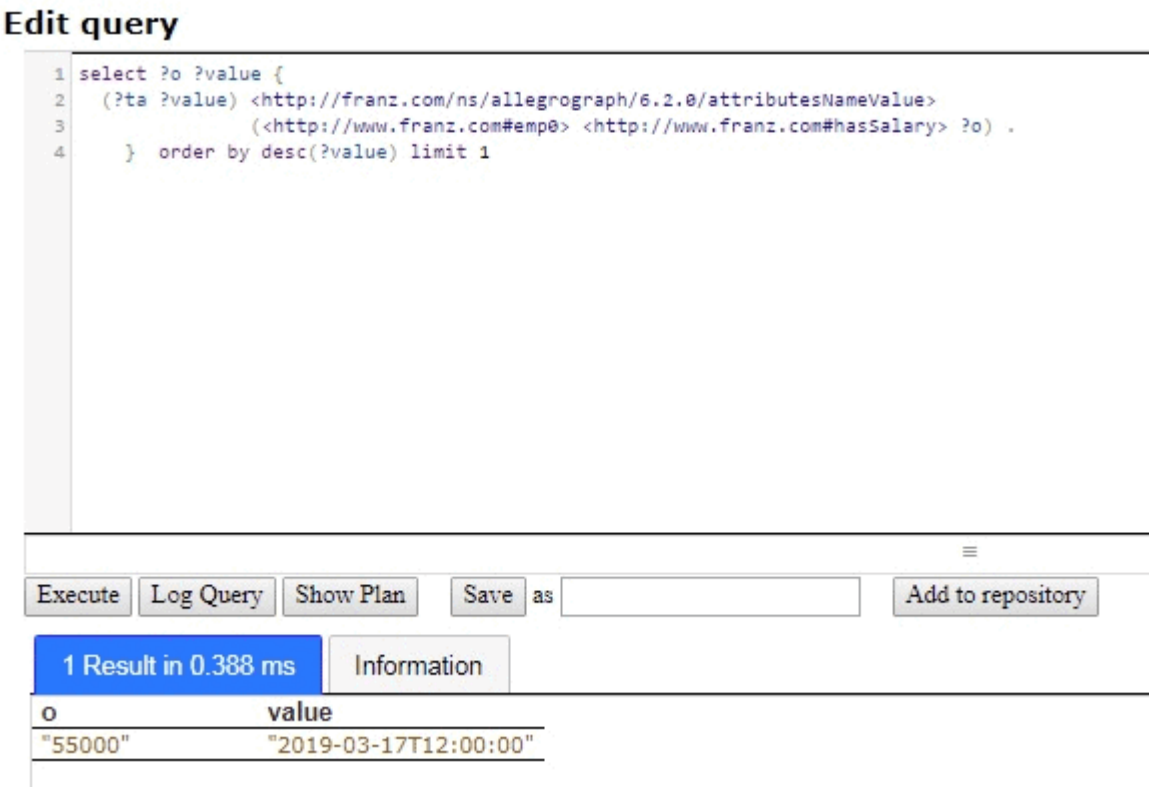
```
salaryChange017 forEmployee emp0
salaryChange017 effectiveDate “2019-01-12T10:12:45”
salaryChange017 oldSalary “50000”
salaryChange017 newSalary “55000”

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” } .
```

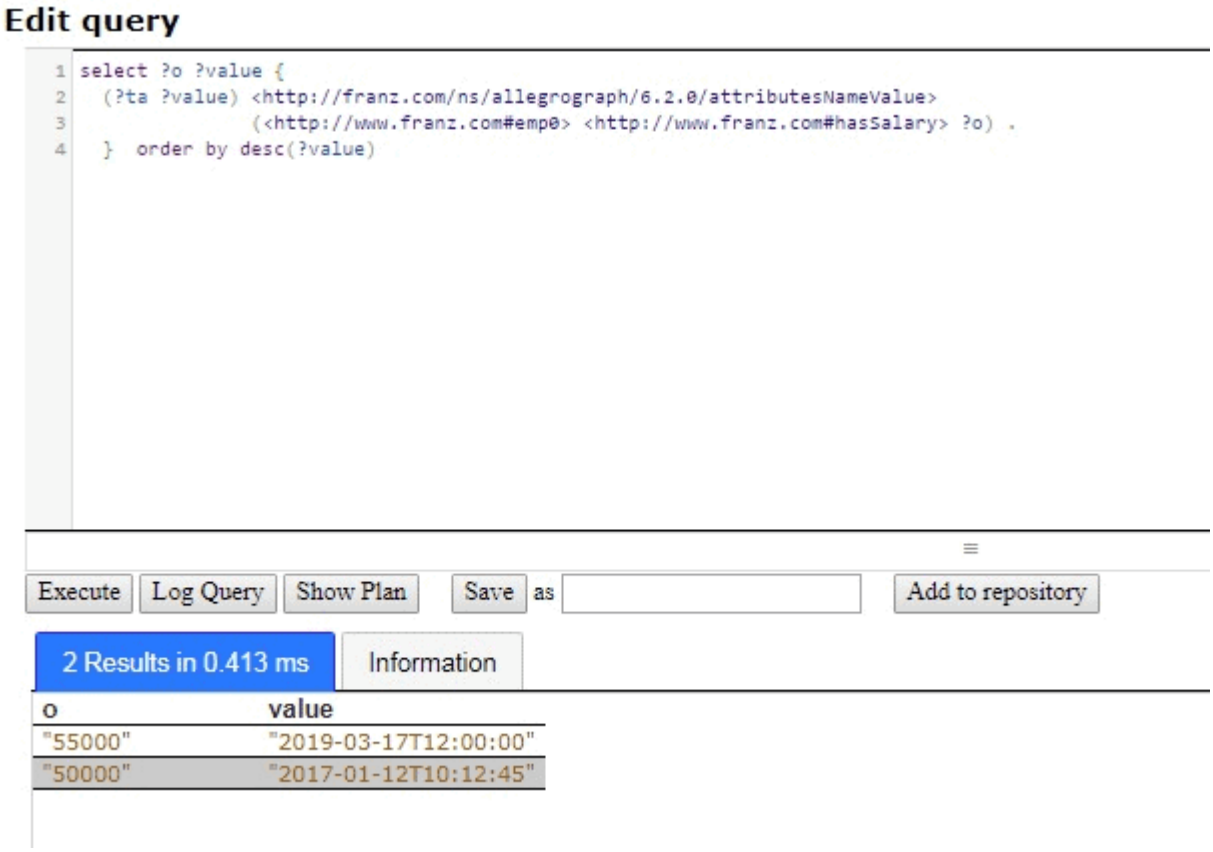
What is the current salary? A simple SPARQL query tells us:

```
 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
```



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)
```



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.