Unraveling the Quandary of Access Layer versus Storage Layer Security

InfoSecurity – February 2019

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.

ГРАФОВЫЕ БАЗЫ: ПРИНЦИП РАБОТЫ И ПРИМЕНЕНИЕ — GRAPH BASES: PRINCIPLE OF OPERATION AND APPLICATION

Всеволод Дёмкин удаленно работает во Franz Inc. над графовой базой AllegroGraph. Преподает в Projector курс «Natural

Language Processing». В свободное время делаетопен-сорс для обработки природных текстов на Lisp'е.

Мы рассмотрим создание программы для агрегации текстов из разных источников, таких как twitter, блоги, reddit и т.д., — их автоматической, а затем ручной обработки для формирования дайджеста новостей по определенной теме. На этом примере мы проанализируем, какие преимущества дает использование графовых баз данных, обсудим их возможности и ограничения.

В качестве конкретной БД будет использована система Franz AllegroGraph и мы ознакомимся с ее экосистемой, включающей возможности построение API и веб-приложений, а также со средой Allegro Common Lisp, на которой она построена. Особое внимание будет уделено использованию машинного обучения и NLP при решении задач работы с текстом, в частности, внутри AllegroGraph.

Обсудим:

— В чем особенности, как работают, преимущества/недостатки графовых БД;

— Как решать базовые задачи обработки текстов с использованием инструментария ML/NLP;

— Как построить полноценное приложение с ядром обработки текста на основе графовой БД и ML/NLP технологий;

— Как устроена экосистема Common Lisp и как можно задействовать ее для создания серверных приложений.

Лекция будет полезна: разработчикам, которые интересуются темой графовых баз данных и/или ML/NLP.

Semantic Web and Semantic Technology Trends in 2019

Dataversity – January 2019

What to expect of Semantic Web and other Semantic Technologies in 2019? Quite a bit. DATAVERSITY engaged with leaders in the space to get their thoughts on how Semantic Technologies will have an impact on multiple areas.

Dr. Jans Aasman, CEO of Franz Inc. was quoted several times in the article:

Among the semantic-driven AI ventures next year will be those that relate to the healthcare space, says Dr. Jans Aasman, CEO of Semantic Web technology company Franz, Inc:

"In the last two years some of the technologies were starting to get used in production," he says. "In 2019 we will see a ramp-up of the number of AI applications that will help save lives by providing early warning signs for impending diseases. Some diseases will be predicted years in advance by using genetic patient data to understand future biological issues, like the likelihood of cancerous mutations – and start preventive therapies before the disease takes hold."

If that's not enough, how about digital immortality via AI Knowledge Graphs, where an interactive voice system will bring public figures in contact with anyone in the real world? "We'll see the first examples of Digital Immortality in 2019 in the form of AI Digital Personas for public figures," says Aasman, whose company is a partner in the Noam Chomsky Knowledge Graph:

"The combination of Artificial Intelligence and

Semantic Knowledge Graphs will be used to transform the works of scientists, technologists, politicians, and scholars like Noam Chomsky into an interactive response system that uses the person's actual voice to answer questions," he comments.

"AI Digital Personas will dynamically link information from various sources – such as books, research papers, notes and media interviews – and turn the disparate information into a knowledge system that people can interact with digitally." These AI Digital Personas could also be used while the person is still alive to broaden the accessibility of their expertise.

On the point of the future of graph visualization apps, Aasman notes that:

"Most graph visualization applications show network diagrams in only two dimensions, but it is unnatural to manipulate graphs on a flat computer screen in 2D. Modern R virtual reality will add at least two dimensions to graph visualization, which will create a more natural way to manipulate complex graphs by incorporating more depth and temporal unfolding to understand information within a time perspective."

Read the full article at Dataversity.

2019 Trends In The Internet

Of Things: The Makings Of An Intelligent IoT

AI Business – December 2018

2019 will be a crucial year for the Internet of Things for two reasons. Firstly, many of the initial predictions for this application of big data prognosticated a future whereby at the start of the next decade there would be billions of connected devices all simultaneously producing sensor data. The IoT is just a year away from making good on those claims.

Dr. Jans Aasman, Franz's CEO was quoted by the author:

The IIoT is the evolution of the IoT that will give it meaning and help it actualize the number of connected devices forecast for the start of the next decade. The IIoT will encompass smart cities, edge devices, wearables, deep learning and classic machine learning alongside lesser acknowledged elements of AI in a basic paradigm in which, according to Franz CEO Jans Aasman, "you can look at the past and learn from certain situations what's likely going to happen. You feed it in your [IoT] system and it does better… then you look at what actually happened and it goes back in your machine learning system. That will be your feedback loop."

Although deep learning relies on many of the same concepts as traditional machine learning, with "deep learning it's just that you do it with more computers and more intermediate layers," Aasman said, which results in higher accuracy levels.

The feedback mechanism described by Aasman has such a tremendous capacity to reform data-driven businesses because of the speed of the iterations provided by low latency IIoT data.

One of the critical learning facets the latter produces

involves optimization, such as determining the best way to optimize route deliveries encompassing a host of factors based on dedicated rules about them. “There’s no way in [Hades] that a machine learning system would be able to do the complex scheduling of 6,000 people,” Aasman declared. “That’s a really complicated thing where you have to think of every factor for every person.”

However, constraint systems utilizing multi-step reasoning can regularly complete such tasks and the optimization activities for smart cities. Aasman commented that for smart cities, semantic inferencing systems can incorporate data from traffic patterns and stop lights, weather predictions, the time of year, and data about specific businesses and their customers to devise rules for optimal event scheduling. Once the events actually take place, their results—as determined by KPIs—can be analyzed with machine learning to issue future predictions about how to better those results in what Aasman called “a beautiful feedback loop between a machine learning system and a rules-based system.”

In almost all of the examples discussed above, the IIoT incorporates cognitive computing “so humans can take action for better business results,” Aasman acknowledged. The means by which these advantages are created are practically limitless.

Read the Full Article at AI Business.

AllegroGraph named to 2019 Trend-Setting Products

Database Trends and Applications – December 2018

You can call it the new oil, or even the new electricity, but however it is described, it's clear that data is now recognized as an essential fuel flowing through organizations and enabling never before seen opportunities. However, data cannot simply be collected; it must be handled with care in order to fulfill the promise of faster, smarter decision making.

More than ever, it is critical to have the right tools for the job. Leading IT vendors are coming forward to help customers address the data-driven possibilities by improving self-service access, real-time insights, governance and security, collaboration, high availability, and more.

To help showcase these innovative products and services each year, Database Trends and Applications magazine looks for offerings that promise to help organizations derive greater benefit from their data, make decisions faster, and work smarter and more securely.

This year our list includes newer approaches leveraging artificial intelligence, machine learning, and automation as well as products in more established categories such as relational and NoSQL database management, MultiValue, performance management, analytics, and data governance.

Read the AllegroGraph Spotlight

AllegroGraph News

Franz periodically distributes newsletters to its Semantic Technologies, and Common Lisp based Enterprise Development Tools mailing lists, providing information on related upcoming events and new software product developments.

Optimizing Fraud Management with AI Knowledge Graphs

From Global Banking and Finance Review – July 12, 2018

This article discusses Knowledge Graphs for Anti-Money Laundering (AML), Suspicious Activity Reports (SAR), counterfeiting and social engineering falsities, as well as synthetic, first-party, and card-not-present fraud.

> *By compiling fraud-related data into an AI knowledge graph, risk management personnel can also triage those alerts for the right action at the right time. They also get the additive benefit of reusing this graph to decrease other risks for security, loans, or additional financial purposes.*

Dr. Aasman goes on to note:

> *By incorporating AI, these threat maps yields a plethora of information for actually preventing fraud. Supervised learning methods can readily identify what events constitute fraud and which don't; many of these involve classic machine*

learning. Unsupervised learning capabilities are influential in determining normal user behavior then pinpointing anomalies contributing to fraud. Perhaps the most effective way AI underpins risk management knowledge graphs is in predicting the likelihood—and when—a specific fraud instance will take place. Once organizations have data for customers, events, and fraud types over a length of time (which could be in as little as a month in the rapidly evolving financial crimes space), they can compute the co-occurrence between events and fraud types.

Read the full article over at Global Banking and Finance Review.

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',
```

```
        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:

```
    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:

```
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:

```
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 »
```