Table of Contents

Introduction 1.1

Modelling Document Inheritance 1.2

Accessing Shapes Data 1.3

AQL 1.4

- Using Joins in AQL 1.4.1
- Using Dynamic Attribute Names 1.4.2
- Creating Test-data using AQL 1.4.3
- Diffing Documents 1.4.4
- Avoiding Parameter Injection 1.4.5
- Multiline Query Strings 1.4.6
- Migrating named graph functions to 3.0 1.4.7
- Migrating anonymous graph functions to 3.0 1.4.8
- Migrating graph measurements to 3.0 1.4.9

Graph 1.5

- Fulldepth Graph-Traversal 1.5.1
- Using a custom Visitor 1.5.2
- Example AQL Queries for Graphs 1.5.3

Use Cases / Examples 1.6

- Crawling Github with Promises 1.6.1
- Using ArangoDB with Sails.js 1.6.2
- Populating a Textbox 1.6.3
- Exporting Data 1.6.4
- Accessing base documents with Java 1.6.5
- Add XML data to ArangoDB with Java 1.6.6

Administration 1.7

- Using Authentication 1.7.1
- Importing Data 1.7.2
- Replicating Data 1.7.3
- XCopy Install Windows 1.7.4
- Migrating 2.8 to 3.0 1.7.5
- Show grants function 1.7.6

Compiling / Build 1.8

- Compile on Debian 1.8.1
- Compile on Windows 1.8.2
- OpenSSL 1.8.3
- Running Custom Build 1.8.4
  - Recompiling jemalloc 1.8.4.1

Cloud, DCOS and Docker 1.9

- Running on AWS 1.9.1
- Update on AWS 1.9.2
- Running on Azure 1.9.3

Docker ArangoDB 1.9.4
Docker with NodeJS App 1.9.5
In the GiantSwarm 1.9.6
ArangoDB in Mesos 1.9.7
DC/OS: Full example 1.9.8
Monitoring 1.10
Collectd - Replication Slaves 1.10.1
Collectd - Network usage 1.10.2
Collectd - more Metrics 1.10.3
Collectd - Monitoring Foxx 1.10.4

Cookbook

This cookbook is filled with recipes to help you understand the multi-model database ArangoDB better and to help you with specific problems.

You can participate and write your own recipes. You only need to write a recipe in markdown and make a pull request to our repository.

Recipes

There will be some simple recipes to bring you closer to ArangoDB and show you the amount of possibilities of our Database. There also will be more complex problems to show you solution to specific problems and the depth of ArangoDB.

Every recipe is divided into three parts:

1. **Problem**: A description of the problem
2. **Solution**: A detailed solution of the given problem with code if any is needed
3. **Comment**: Explanation of the solution. This part is optional depending on the complexity of the problem

Every recipe has tags to for a better overview:

#api, *#aql*, *#arangosh*, *#collection*, *#database*, *#debian*, *#docker*, *#document*, *#driver*, *#foxx*, *#giantswarm*, *#graph*, *#howto*, *#java*, *#javascript*, *#join*, *#nodejs*, *#windows*

Model document inheritance

Problem

How do you model document inheritance given that collections do not support that feature?

Solution

Lets assume you have three document collections: "subclass", "class" and "superclass". You also have two edge collections: "sub_extends_class" and "class_extends_super".

You can create them via arangosh or foxx:

```
var graph_module = require("com/arangodb/general-graph");
var g = graph_module._create("inheritance");
g._extendEdgeDefinitions(graph_module. _directedRelation("sub_extends_class", ["subclass"], ["class"]));
g._extendEdgeDefinitions(graph_module. _directedRelation("class_extends_super", ["class"], ["superclass"]));
```

This makes sure when using the graph interface that the inheritance looks like:

- sub → class
- class → super
- super → sub

To make sure everything works as expected you should use the built-in traversal in combination with Foxx. This allows you to add the inheritance security layer easily. To use traversals in foxx simply add the following line before defining routes:

```
var traversal = require("org/arangodb/graph/traversal");
var Traverser = traversal.Traverser;
```

Also you can add the following endpoint in Foxx:

```
var readerConfig = {
  datasource: traversal.graphDatasourceFactory("inheritance"),
  expander: traversal.outboundExpander, // Go upwards in the tree
  visitor: function (config, result, vertex, path) {
    for (key in vertex) {
      if (vertex.hasOwnProperty(key) && !result.hasOwnProperty(key)) {
        result[key] = vertex[key] // Store only attributes that have not yet been found
      }
    }
  }
};

controller.get("load/:collection/:key", function(req, res) {
  var result = {};
  var id = res.params("collection") + "/" + res.params("key");
  var traverser = new Traverser(readerConfig);
  traverser.traverse(result, g.getVertex(id));
  res.json(result);
});
```

This will make sure to iterate the complete inheritance tree upwards to the root element and will return all values on the path were the first instance of this value is kept

Comment

You should go with edges because it is much easier to query them if you have a theoretically unlimited depth in inheritance. If you have a fixed inheritance depth you could also go with an attribute in the document referencing the parent and execute joins in AQL.

Author: Michael Hackstein

Tags: #graph #document

Accessing Shapes Data

Problem

Documents in a collection may have different shapes associated with them. There is no way to query the shapes data directly. So how do you solve this problem?

Solution

There are two possible ways to do this.

A) The fast way with some random samplings:

1. Ask for a random document (`db.<collection>.any()`) and note its top-level attribute names
2. Repeat this for at least 10 times. After that repeat it only if you think it's worth it.

Following is an example of an implementation:

```
attributes(db.myCollection);


function attributes(collection) {
  "use strict"

  var probes = 10;
  var maxRounds = 3;
  var threshold = 0.5;

  var maxDocuments = collection.count();

  if (maxDocuments < probes) {
    probes = maxDocuments;
  }

  if (probes === 0) {
    return [ ];
  }

  var attributes = { };

  while (maxRounds--) {
    var newDocuments = 0;
    var n = probes;
    while (n--) {
      var doc = collection.any();
      var found = false;
      var keys = Object.keys(doc);

    for (var i = 0; i < keys.length; ++i) {
      if (attributes.hasOwnProperty(keys[i])) {
        ++attributes[keys[i]];
      }
      else {
        attributes[keys[i]] = 1;
        found = true;
      }
    }

     if (found) {
      ++newDocuments;
     }
    }

    if (newDocuments / probes <= threshold) {
      break;
    }
  }
```

```
  return Object.keys(attributes);
}
```

B) The way to find all top-level attributes

If you don't mind to make some extra inserts and you don't care about deletion or updates of documents you can use the following:

```
db._create("mykeys");
db.mykeys.ensureUniqueSkiplist("attribute");


function insert(collection, document) {
  var result = collection.save(document);

  try {
    var keys = Objects.keys(document);

    for (i = 0; i < keys.length; ++i) {
      try {
        db.mykeys.save({ attribute: keys[i] });
      }
        catch (err1) {
        // potential unique key constraint violations
        }
    }
  }
  catch (err2) {
  }

  return result;
}
```

Comment

A) The fast way with some random samplings:

You get some random sampling with bounded complexity. If you have a variety of attributes you should repeat the procedure more than 10 times.

The procedure can be implemented as a server side action.

B) The way to find all top-level attributes:

This procedure will not care about updates or deletions of documents. Also only the top-level attribute of the documents will be inserted and nested one ignored.

The procedure can be implemented as a server side action.

Author: Arangodb

Tags: #collection #database

AQL

Using AQL in general

- Using Joins in AQL
- Using Dynamic Attribute Names
- Creating Test-data using AQL
- Diffing Documents
- Avoiding Parameter Injection
- Multiline Query Strings

Migrating from 2.x to 3.0

- Migrating named graph functions to 3.0
- Migrating anonymous graph functions to 3.0
- Migrating graph measurements to 3.0

Using Joins in AQL

Problem

I want to join documents from collections in an AQL query.

- One-to-Many: I have a collection users and a collection cities. A user lives in a city and I need the city information during the query.
- Many-To-Many: I have a collection authors and books. An author can write many books and a book can have many authors. I want to return a list of books with their authors. Therefore I need to join the authors and books.

Solution

Unlike many NoSQL databases, ArangoDB does support joins in AQL queries. This is similar to the way traditional relational databases handle this. However, because documents allow for more flexibility, joins are also more flexible. The following sections provide solutions for common questions.

One-To-Many

You have a collection called users. Users live in city and a city is identified by its primary key. In principle you can embedded the city document into the users document and be happy with it.

```
{
  "_id" : "users/2151975421",
  "_key" : "2151975421",
  "_rev" : "2151975421",
  "name" : {
    "first" : "John",
    "last" : "Doe"
  },
  "city" : {
    "name" : "Metropolis"
  }
}
```

This works well for many use cases. Now assume, that you have additional information about the city, like the number of people living in it. It would be impractical to change each and every user document if this numbers changes. Therefore it is good idea to hold the city information in a separate collection.

```
arangosh> db.cities.document("cities/2241300989");
{
  "population" : 1000,
  "name" : "Metropolis",
  "_id" : "cities/2241300989",
  "_rev" : "2241300989",
  "_key" : "2241300989"
}
```

Now you instead of embedding the city directly in the user document, you can use the key of the city.

```
arangosh> db.users.document("users/2290649597");
{
  "name" : {
    "first" : "John",
    "last" : "Doe"
  },
  "city" : "cities/2241300989",
  "_id" : "users/2290649597",
  "_rev" : "2290649597",
  "_key" : "2290649597"
}
```

We can now join these two collections very easily.

```
arangosh> db._query(
........>"FOR u IN users " +
........>"  FOR c IN cities " +
........>"    FILTER u.city == c._id RETURN { user: u, city: c }"
........>).toArray()
[
  {
    "user" : {
      "name" : {
        "first" : "John",
        "last" : "Doe"
      },
      "city" : "cities/2241300989",
      "_id" : "users/2290649597",
      "_rev" : "2290649597",
      "_key" : "2290649597"
    },
    "city" : {
      "population" : 1000,
      "name" : "Metropolis",
      "_id" : "cities/2241300989",
      "_rev" : "2241300989",
      "_key" : "2241300989"
    }
  }
]
```

Unlike SQL there is no special JOIN keyword. The optimizer ensures that the primary index is used in the above query.

However, very often it is much more convenient for the client of the query if a single document would be returned, where the city information is embedded in the user document - as in the simple example above. With AQL there you do not need to forgo this simplification.

```
arangosh> db._query(
........>"FOR u IN users " +
........>"  FOR c IN cities " +
........>"    FILTER u.city == c._id RETURN merge(u, {city: c})"
........>).toArray()
[
  {
    "_id" : "users/2290649597",
    "_key" : "2290649597",
    "_rev" : "2290649597",
    "name" : {
      "first" : "John",
      "last" : "Doe"
    },
    "city" : {
      "_id" : "cities/2241300989",
      "_key" : "2241300989",
      "_rev" : "2241300989",
      "population" : 1000,
      "name" : "Metropolis"
    }
  }
]
```

So you can have both: the convenient representation of the result for your client and the flexibility of joins for your data model.

Many-To-Many

In the relational word you need a third table to model the many-to-many relation. In ArangoDB you have a choice depending on the information you are going to store and the type of questions you are going to ask.

Assume that authors are stored in one collection and books in a second. If all you need is "which are the authors of a book" then you can easily model this as a list attribute in users.

If you want to store more information, for example which author wrote which page in a conference proceeding, or if you also want to know "which books were written by which author", you can use edge collections. This is very similar to the "join table" from the relational world.

Embedded Lists

If you only want to store the authors of a book, you can embed them as list in the book document. There is no need for a separate collection.

```
arangosh> db.authors.toArray()
[
  {
    "_id" : "authors/2661190141",
    "_key" : "2661190141",
    "_rev" : "2661190141",
    "name" : {
      "first" : "Maxima",
      "last" : "Musterfrau"
    }
  },
  {
    "_id" : "authors/2658437629",
    "_key" : "2658437629",
    "_rev" : "2658437629",
    "name" : {
      "first" : "John",
      "last" : "Doe"
    }
  }
]
```

You can query books

```
arangosh> db._query("FOR b IN books RETURN b").toArray();
[
  {
    "_id" : "books/2681506301",
    "_key" : "2681506301",
    "_rev" : "2681506301",
    "title" : "The beauty of JOINS",
    "authors" : [
      "authors/2661190141",
      "authors/2658437629"
    ]
  }
]
```

and join the authors in a very similar manner given in the one-to-many section.

```
arangosh> db._query(
........>"FOR b IN books " +
........>"  LET a = (FOR x IN b.authors " +
........>"             FOR a IN authors FILTER x == a._id RETURN a) " +
........>"   RETURN { book: b, authors: a }"
........>).toArray();
[
  {
    "book" : {
      "title" : "The beauty of JOINS",
      "authors" : [
        "authors/2661190141",
        "authors/2658437629"
      ],
      "_id" : "books/2681506301",
      "_rev" : "2681506301",
      "_key" : "2681506301"
    },
    "authors" : [
      {
        "name" : {
```

```
          "first" : "Maxima",
          "last" : "Musterfrau"
        },
        "_id" : "authors/2661190141",
        "_rev" : "2661190141",
        "_key" : "2661190141"
      },
      {
        "name" : {
          "first" : "John",
          "last" : "Doe"
        },
        "_id" : "authors/2658437629",
        "_rev" : "2658437629",
        "_key" : "2658437629"
      }
    ]
  }
]
```

or embed the authors directly

```
arangosh> db._query(
........>"FOR b IN books LET a = (" +
........>"     FOR x IN b.authors " +
........>"        FOR a IN authors FILTER x == a._id RETURN a)" +
........>"  RETURN merge(b, { authors: a })"
........>).toArray();
[
  {
    "_id" : "books/2681506301",
    "_key" : "2681506301",
    "_rev" : "2681506301",
    "title" : "The beauty of JOINS",
    "authors" : [
      {
        "_id" : "authors/2661190141",
        "_key" : "2661190141",
        "_rev" : "2661190141",
        "name" : {
          "first" : "Maxima",
          "last" : "Musterfrau"
        }
      },
      {
        "_id" : "authors/2658437629",
        "_key" : "2658437629",
        "_rev" : "2658437629",
        "name" : {
          "first" : "John",
          "last" : "Doe"
        }
      }
    ]
  }
]
```

Using Edge Collections

If you also want to query which books are written by a given author, embedding authors in the book document is possible, but it is more efficient to use a edge collections for speed.

Or you are publishing a proceeding, then you want to store the pages the author has written as well. This information can be stored in the edge document.

First create the users

```
arangosh> db._create("authors");
[ArangoCollection 2926807549, "authors" (type document, status loaded)]

arangosh> db.authors.save({ name: { first: "John", last: "Doe" } })
```

```
{
  "error" : false,
  "_id" : "authors/2935261693",
  "_rev" : "2935261693",
  "_key" : "2935261693"
}

arangosh> db.authors.save({ name: { first: "Maxima", last: "Musterfrau" } })
{
  "error" : false,
  "_id" : "authors/2938210813",
  "_rev" : "2938210813",
  "_key" : "2938210813"
}
```

Now create the books without any author information.

```
arangosh> db._create("books");
[ArangoCollection 2928380413, "books" (type document, status loaded)]

arangosh> db.books.save({ title: "The beauty of JOINS" });
{
  "error" : false,
  "_id" : "books/2980088317",
  "_rev" : "2980088317",
  "_key" : "2980088317"
}
```

An edge collection is now used to link authors and books.

```
arangosh> db._createEdgeCollection("written");
[ArangoCollection 2931132925, "written" (type edge, status loaded)]

arangosh> db.written.save("authors/2935261693",
........>"books/2980088317",
........>{ pages: "1-10" })
{
  "error" : false,
  "_id" : "written/3006237181",
  "_rev" : "3006237181",
  "_key" : "3006237181"
}

arangosh> db.written.save("authors/2938210813",
........>"books/2980088317",
........>{ pages: "11-20" })
{
  "error" : false,
  "_id" : "written/3012856317",
  "_rev" : "3012856317",
  "_key" : "3012856317"
}
```

In order to get all books with their authors you can use NEIGHBORS.

```
arangosh> db._query(
........>"FOR b IN books RETURN " +
........>"  { " +
........>"    book: b, " +
........>"    authors: NEIGHBORS(books," +
........>"                       written," +
........>"                       b._id," +
........>"                      'inbound'" +
........>" ) }"
........>).toArray();
[
  {
    "book" : {
      "_id" : "books/2980088317",
      "_rev" : "2980088317",
      "_key" : "2980088317",
      "title" : "The beauty of JOINS"
```

```
    },
    "authors" : [
      {
        "edge" : {
          "_id" : "written/3006237181",
          "_from" : "authors/2935261693",
          "_to" : "books/2980088317",
          "_rev" : "3006237181",
          "_key" : "3006237181",
          "pages" : "1-10"
        },
        "vertex" : {
          "_id" : "authors/2935261693",
          "_rev" : "2935261693",
          "_key" : "2935261693",
          "name" : {
            "first" : "John",
            "last" : "Doe"
          }
        }
      },
      {
        "edge" : {
          "_id" : "written/3012856317",
          "_from" : "authors/2938210813",
          "_to" : "books/2980088317",
          "_rev" : "3012856317",
          "_key" : "3012856317",
          "pages" : "11-20"
        },
        "vertex" : {
          "_id" : "authors/2938210813",
          "_rev" : "2938210813",
          "_key" : "2938210813",
          "name" : {
            "first" : "Maxima",
            "last" : "Musterfrau"
          }
        }
      }
    ]
  }
]
```

Or if you want to hide the information stored in the edge.

```
arangosh> db._query(
........>"FOR b IN books RETURN {" +
........>"     book: b, authors: " +
........>"          NEIGHBORS(books, written, b._id, 'inbound')[*].vertex }"
........>).toArray();
[
  {
    "book" : {
      "title" : "The beauty of JOINS",
      "_id" : "books/2980088317",
      "_rev" : "2980088317",
      "_key" : "2980088317"
    },
    "authors" : [
      {
        "_id" : "authors/2935261693",
        "_rev" : "2935261693",
        "_key" : "2935261693",
        "name" : {
          "first" : "John",
          "last" : "Doe"
        }
      },
      {
        "_id" : "authors/2938210813",
        "_rev" : "2938210813",
        "_key" : "2938210813",
        "name" : {
          "first" : "Maxima",
```

```
          "last" : "Musterfrau"
        }
      }
    ]
  }
]
```

Or again embed the authors directly into the book document.

```
arangosh> db._query(
........>"FOR b IN books RETURN merge(" +
........>"      b, " +
........>"        {" +
........>"          authors:" +
........>"          NEIGHBORS(books, written, b._id, 'inbound')[*].vertex})"
........>).toArray();
[
  {
    "_id" : "books/2980088317",
    "_rev" : "2980088317",
    "_key" : "2980088317",
    "title" : "The beauty of JOINS",
    "authors" : [
      {
        "_id" : "authors/2935261693",
        "_rev" : "2935261693",
        "_key" : "2935261693",
        "name" : {
          "first" : "John",
          "last" : "Doe"
        }
      },
      {
        "_id" : "authors/2938210813",
        "_rev" : "2938210813",
        "_key" : "2938210813",
        "name" : {
          "first" : "Maxima",
          "last" : "Musterfrau"
        }
      }
    ]
  }
]
```

If you need the authors and their books, simply reverse the direction.

```
arangosh> db._query(
........>"FOR a IN authors RETURN " +
........>"  merge(a, " +
........>"        {books: NEIGHBORS(authors, written, a._id, 'outbound')[*].vertex})"
........>).toArray();
[
  {
    "_id" : "authors/2938210813",
    "_rev" : "2938210813",
    "_key" : "2938210813",
    "name" : {
      "first" : "Maxima",
      "last" : "Musterfrau"
    },
    "books" : [
      {
        "_id" : "books/2980088317",
        "_rev" : "2980088317",
        "_key" : "2980088317",
        "title" : "The beauty of JOINS"
      }
    ]
  },
  {
    "_id" : "authors/2935261693",
    "_rev" : "2935261693",
```

```
    "_key" : "2935261693",
    "name" : {
      "first" : "John",
      "last" : "Doe"
    },
    "books" : [
      {
        "_id" : "books/2980088317",
        "_rev" : "2980088317",
        "_key" : "2980088317",
        "title" : "The beauty of JOINS"
      }
    ]
  }
]
```

Authors: Frank Celler

Tags: #join #aql

Using dynamic attribute names in AQL

Problem

I want an AQL query to return results with attribute names assembled by a function, or with a variable number of attributes.

This will not work by specifying the result using a regular object literal, as object literals require the names and numbers of attributes to be fixed at query compile time.

Solution

There are several solutions to getting dynamic attribute names to work.

Subquery solution

A general solution is to let a subquery or another function to produce the dynamic attribute names, and finally pass them through the `ZIP()` function to create an object from them.

Let's assume we want to process the following input documents:

```
{ "name" : "test", "gender" : "f", "status" : "active", "type" : "user" }
{ "name" : "dummy", "gender" : "m", "status" : "inactive", "type" : "unknown", "magicFlag" : 23 }
```

Let's also assume our goal for each of these documents is to return only the attribute names that contain the letter `a`, together with their respective values.

To extract the attribute names and values from the original documents, we can use a subquery as follows:

```
LET documents = [
  { "name" : "test"," gender" : "f", "status" : "active", "type" : "user" },
  { "name" : "dummy", "gender" : "m", "status" : "inactive", "type" : "unknown", "magicFlag" : 23 }
]

FOR doc IN documents
  RETURN (
    FOR name IN ATTRIBUTES(doc)
      FILTER LIKE(name, '%a%')
      RETURN {
        name: name,
        value: doc[name]
      }
  )
```

The subquery will only let attribute names pass that contain the letter `a`. The results of the subquery are then made available to the main query and will be returned. But the attribute names in the result are still `name` and `value`, so we're not there yet.

So let's also employ AQL's `ZIP()` function, which can create an object from two arrays:

- the first parameter to `ZIP()` is an array with the attribute names
- the second parameter to `ZIP()` is an array with the attribute values

Instead of directly returning the subquery result, we first capture it in a variable, and pass the variable's `name` and `value` components into `ZIP()` like this:

```
LET documents = [
  { "name" : "test"," gender" : "f", "status" : "active", "type" : "user" },
  { "name" : "dummy", "gender" : "m", "status" : "inactive", "type" : "unknown", "magicFlag" : 23 }
]

FOR doc IN documents
  LET attributes = (
```

```
    FOR name IN ATTRIBUTES(doc)
      FILTER LIKE(name, '%a%')
      RETURN {
        name: name,
        value: doc[name]
      }
  )
  RETURN ZIP(attributes[*].name, attributes[*].value)
```

Note that we have to use the expansion operator (`[*]`) on `attributes` because `attributes` itself is an array, and we want either the `name` attribute or the `value` attribute of each of its members.

To prove this is working, here is the above query's result:

```
[
  {
    "name": "test",
    "status": "active"
  },
  {
    "name": "dummy",
    "status": "inactive",
    "magicFlag": 23
  }
]
```

As can be seen, the two results have a different amount of result attributes. We can also make the result a bit more dynamic by prefixing each attribute with the value of the `name` attribute:

```
LET documents = [
  { "name" : "test"," gender" : "f", "status" : "active", "type" : "user" },
  { "name" : "dummy", "gender" : "m", "status" : "inactive", "type" : "unknown", "magicFlag" : 23 }
]

FOR doc IN documents
  LET attributes = (
    FOR name IN ATTRIBUTES(doc)
      FILTER LIKE(name, '%a%')
      RETURN {
        name: CONCAT(doc.name, '-', name),
        value: doc[name]
      }
  )
  RETURN ZIP(attributes[*].name, attributes[*].value)
```

That will give us document-specific attribute names like this:

```
[
  {
    "test-name": "test",
    "test-status": "active"
  },
  {
    "dummy-name": "dummy",
    "dummy-status": "inactive",
    "dummy-magicFlag": 23
  }
]
```

Using expressions as attribute names (ArangoDB 2.5)

If the number of dynamic attributes to return is known in advance, and only the attribute names need to be calculated using an expression, then there is another solution.

ArangoDB 2.5 and higher allow using expressions instead of fixed attribute names in object literals. Using expressions as attribute names requires enclosing the expression in extra `[` and `]` to disambiguate them from regular, unquoted attribute names.

Let's create a result that returns the original document data contained in a dynamically named attribute. We'll be using the expression `doc.type` for the attribute name. We'll also return some other attributes from the original documents, but prefix them with the documents' `_key` attribute values. For this we also need attribute name expressions.

Here is a query showing how to do this. The attribute name expressions all required to be enclosed in `[` and `]` in order to make this work:

```
LET documents = [
  { "_key" : "3231748397810", "gender" : "f", "status" : "active", "type" : "user" },
  { "_key" : "3231754427122", "gender" : "m", "status" : "inactive", "type" : "unknown" }
]

FOR doc IN documents
  RETURN {
    [ doc.type ] : {
      [ CONCAT(doc._key, "_gender") ] : doc.gender,
      [ CONCAT(doc._key, "_status") ] : doc.status
    }
  }
```

This will return:

```
[
  {
    "user": {
      "3231748397810_gender": "f",
      "3231748397810_status": "active"
    }
  },
  {
    "unknown": {
      "3231754427122_gender": "m",
      "3231754427122_status": "inactive"
    }
  }
]
```

Note: attribute name expressions and regular, unquoted attribute names can be mixed.

Author: Jan Steemann

Tags: #aql

Creating test data with AQL

Problem

I want to create some test documents.

Solution

If you haven't yet created a collection to hold the documents, create one now using the ArangoShell:

```
db._create("myCollection");
```

This has created a collection named *myCollection*.

One of the easiest ways to fill a collection with test data is to use an AQL query that iterates over a range.

Run the following AQL query from the **AQL editor** in the web interface to insert 1,000 documents into the just created collection:

```
FOR i IN 1..1000
  INSERT { name: CONCAT("test", i) } IN myCollection
```

The number of documents to create can be modified easily be adjusting the range boundary values.

To create more complex test data, adjust the AQL query!

Let's say we also want a `status` attribute, and fill it with integer values between `1` to (including) `5`, with equal distribution. A good way to achieve this is to use the modulo operator (`%`):

```
FOR i IN 1..1000
  INSERT {
    name: CONCAT("test", i),
    status: 1 + (i % 5)
  } IN myCollection
```

To create pseudo-random values, use the `RAND()` function. It creates pseudo-random numbers between 0 and 1. Use some factor to scale the random numbers, and `FLOOR()` to convert the scaled number back to an integer.

For example, the following query populates the `value` attribute with numbers between 100 and 150 (including):

```
FOR i IN 1..1000
  INSERT {
    name: CONCAT("test", i),
    value: 100 + FLOOR(RAND() * (150 - 100 + 1))
  } IN myCollection
```

After the test data has been created, it is often helpful to verify it. The `RAND()` function is also a good candidate for retrieving a random sample of the documents in the collection. This query will retrieve 10 random documents:

```
FOR doc IN myCollection
  SORT RAND()
  LIMIT 10
  RETURN doc
```

The `COLLECT` clause is an easy mechanism to run an aggregate analysis on some attribute. Let's say we wanted to verify the data distribution inside the `status` attribute. In this case we could run:

```
FOR doc IN myCollection
  COLLECT value = doc.value WITH COUNT INTO count
  RETURN {
```

```
    value: value,
    count: count
  }
```

The above query will provide the number of documents per distinct `value`.

Author: Jan Steemann

Tags: #aql

Diffing Two Documents in AQL

Problem

How to create a `diff` of documents in AQL

Solution

Though there is no built-in AQL function to `diff` two documents, it is easily possible to build your own like in the following query:

```
/* input document 1*/
LET doc1 = {
  "foo" : "bar",
  "a" : 1,
  "b" : 2
}

/* input document 2 */
LET doc2 = {
  "foo" : "baz",
  "a" : 2,
  "c" : 3
}

/* collect attributes present in doc1, but missing in doc2 */
LET missing = (
  FOR key IN ATTRIBUTES(doc1)
  FILTER ! HAS(doc2, key)
  RETURN {
    [ key ]: doc1[key]
  }
)

/* collect attributes present in both docs, but that have different values */
LET changed = (
  FOR key IN ATTRIBUTES(doc1)
    FILTER HAS(doc2, key) && doc1[key] != doc2[key]
    RETURN {
      [ key ] : {
        old: doc1[key],
        new: doc2[key]
      }
    }
)

/* collect attributes present in doc2, but missing in doc1 */
LET added = (
  FOR key IN ATTRIBUTES(doc2)
    FILTER ! HAS(doc1, key)
    RETURN {
      [ key ] : doc2[key]
    }
)

/* return final result */
RETURN {
  "missing" : missing,
  "changed" : changed,
  "added" : added
}
```

Note: The query may look a bit lengthy, but much of that is due to formatting. A more terse version can be found below.

The above query will return a document with three attributes:

- *missing*: Contains all attributes only present in first document (i.e. missing in second document)

- *changed*: Contains all attributes present in both documents that have different values
- *added*: Contains all attributes only present in second document (i.e. missing in first document)

For the two example documents it will return:

```
[
 {
   "missing" : [
     {
       "b" : 2
     }
   ],
   "changed" : [
     {
       "foo" : {
         "old" : "bar",
         "new" : "baz"
       }
      },
     {
       "a" : {
         "old" : 1,
         "new" : 2
       }
     }
   ],
   "added" : [
     {
       "c" : 3
     }
   ]
 }
]
```

That output format was the first that came to my mind. It is of course possible to adjust the query so it produces a different output format.

Following is a version of the same query that can be invoked from JavaScript easily. It passes the two documents as bind parameters and calls `db._query`. The query is now an one-liner (less readable but easier to copy&paste):

```
bindVariables = {
  doc1 : { "foo" : "bar", "a" : 1, "b" : 2 },
  doc2 : { "foo" : "baz", "a" : 2, "c" : 3 }
};

query = "LET doc1 = @doc1, doc2 = @doc2, missing = (FOR key IN ATTRIBUTES(doc1) FILTER ! HAS(doc2, key) RETURN { [ key ]: doc1[
key] }), changed = (FOR key IN ATTRIBUTES(doc1) FILTER HAS(doc2, key) && doc1[key] != doc2[key] RETURN { [ key ] : { old: doc1[
key], new: doc2[key] } }), added = (FOR key IN ATTRIBUTES(doc2) FILTER ! HAS(doc1, key) RETURN { [ key ] : doc2[key] }) RETURN
{ missing : missing, changed : changed, added : added }";

result = db._query(query, bindVariables).toArray();
```

Author: Jan Steemann

Tags: #howto #aql

Avoiding parameter injection in AQL

Problem

I don't want my AQL queries to be affected by parameter injection.

What is parameter injection?

Parameter injection means that potentially content is inserted into a query, and that injection may change the meaning of the query. It is a security issue that may allow an attacker to execute arbitrary queries on the database data.

It often occurs if applications trustfully insert user-provided inputs into a query string, and do not fully or incorrectly filter them. It also occurs often when applications build queries naively, without using security mechanisms often provided by database software or querying mechanisms.

Parameter injection examples

Assembling query strings with simple string concatenation looks trivial, but is potentially unsafe. Let's start with a simple query that's fed with some dynamic input value, let's say from a web form. A client application or a Foxx route happily picks up the input value, and puts it into a query:

```
/* evil ! */
var what = req.params("searchValue");  /* user input value from web form */
...
var query = "FOR doc IN collection FILTER doc.value == " + what + " RETURN doc";
db._query(query, params).toArray();
```

The above will probably work fine for numeric input values.

What could an attacker do to this query? Here are a few suggestions to use for the `searchValue` parameter:

- for returning all documents in the collection: `1 || true`
- for removing all documents: `1 || true REMOVE doc IN collection //`
- for inserting new documents: `1 || true INSERT { foo: "bar" } IN collection //`

It should have become obvious that this is extremely unsafe and should be avoided.

An pattern often seen to counteract this is trying to quote and escape potentially unsafe input values before putting them into query strings. This may work in some situations, but it's easy to overlook something or get it subtly wrong:

```
/* we're sanitzing now, but it's still evil ! */
var value = req.params("searchValue").replace(/'/g, '');
...
var query = "FOR doc IN collection FILTER doc.value == '" + value + "' RETURN doc";
db._query(query, params).toArray();
```

The above example uses single quotes for enclosing the potentially unsafe user input, and also replaces all single quotes in the input value beforehand. Not only may that change the user input (leading to subtle errors such as *"why does my search for `O'Brien` don't return any results?"*), but it is also unsafe. If the user input contains a backslash at the end (e.g. `foo bar\`), that backslash will escape the closing single quote, allowing the user input to break out of the string fence again.

It gets worse if user input is inserted into the query at multiple places. Let's assume we have a query with two dynamic values:

```
query = "FOR doc IN collection FILTER doc.value == '" + value + "' && doc.type == '" + type + "' RETURN doc";
```

If an attacker inserted `\` for parameter `value` and `|| true REMOVE doc IN collection //` for parameter `type` , then the effective query would become

```
FOR doc IN collection FILTER doc.value == '\' && doc.type == ' || true REMOVE doc IN collection //' RETURN doc
```

which is highly undesirable.

Solution

Instead of mixing query string fragments with user inputs naively via string concatenation, use either **bind parameters** or a **query builder**. Both can help to avoid the problem of injection, because they allow separating the actual query operations (like `FOR`, `INSERT`, `REMOVE`) from (user input) values.

This recipe focuses on using bind parameters. This is not to say that query builders shouldn't be used. They were simply omitted here for the sake of simplicity. To get started with a using an AQL query builder in ArangoDB or other JavaScript environments, have a look at aqb (which comes bundled with ArangoDB). Inside ArangoDB, there are also Foxx queries which can be combined with aqb.

What bind parameters are

Bind parameters in AQL queries are special tokens that act as placeholders for actual values. Here's an example:

```
FOR doc IN collection
  FILTER doc.value == @what
  RETURN doc
```

In the above query, `@what` is a bind parameter. In order to execute this query, a value for bind parameter `@what` must be specified. Otherwise query execution will fail with error 1551 (*no value specified for declared bind parameter*). If a value for `@what` gets specified, the query can be executed. However, the query string and the bind parameter values (i.e. the contents of the `@what` bind parameter) will be handled separately. What's in the bind parameter will always be treated as a value, and it can't get out of its sandbox and change the semantic meaning of a query.

How bind parameters are used

To execute a query with bind parameters, the query string (containing the bind parameters) and the bind parameter values are specified separately (note that when the bind parameter value is assigned, the prefix `@` needs to be omitted):

```
/* query string with bind parameter */
var query = "FOR doc IN collection FILTER doc.value == @what RETURN doc";

/* actual value for bind parameter */
var params = { what: 42 };

/* run query, specifying query string and bind parameter separately */
db._query(query, params).toArray();
```

If a malicious user would set `@what` to a value of `1 || true`, this wouldn't do any harm. AQL would treat the contents of `@what` as a single string token, and the meaning of the query would remain unchanged. The actually executed query would be:

```
FOR doc IN collection
  FILTER doc.value == "1 || true"
  RETURN doc
```

Thanks to bind parameters it is also impossible to turn a selection (i.e. read-only) query into a data deletion query.

Using JavaScript variables as bind parameters

There is also a template string generator function `aql` that can be used to safely (and conveniently) built AQL queries using JavaScript variables and expressions. It can be invoked as follows:

```
const aql = require('@arangodb')aql; // not needed in arangosh

var value = "some input value";
```

```
var query = aql`FOR doc IN collection
  FILTER doc.value == ${value}
  RETURN doc`;
var result = db._query(query).toArray();
```

Note that an ES6 template string is used for populating the `query` variable. The string is assembled using the `aql` generator function which is bundled with ArangoDB. The template string can contain references to JavaScript variables or expressions via `${...}`. In the above example, the query references a variable named `value`. The `aql` function generates an object with two separate attributes: the query string, containing references to bind parameters, and the actual bind parameter values.

Bind parameter names are automatically generated by the `aql` function:

```
var value = "some input value";
aql`FOR doc IN collection FILTER doc.value == ${value} RETURN doc`;

{
  "query" : "FOR doc IN collection FILTER doc.value == @value0 RETURN doc",
  "bindVars" : {
    "value0" : "some input value"
  }
}
```

Using bind parameters in dynamic queries

Bind parameters are helpful, so it makes sense to use them for handling the dynamic values. You can even use them for queries that itself are highly dynamic, for example with conditional `FILTER` and `LIMIT` parts. Here's how to do this:

```
/* note: this example has a slight issue... hang on reading */
var query = "FOR doc IN collection";
var params = { };

if (useFilter) {
  query += " FILTER doc.value == @what";
  params.what = req.params("searchValue");
}

if (useLimit) {
  /* not quite right, see below */
  query += " LIMIT @offset, @count";
  params.offset = req.params("offset");
  params.count = req.params("count");
}

query += " RETURN doc";
db._query(query, params).toArray();
```

Note that in this example we're back to string concatenation, but without the problem of the query being vulnerable to arbitrary modifications.

Input value validation and sanitation

Still you should prefer to be paranoid, and try to detect invalid input values as early as possible, at least before executing a query with them. This is because some input parameters may affect the runtime behavior of queries negatively or, when modified, may lead to queries throwing runtime errors instead of returning valid results. This isn't something an attacker should deserve.

`LIMIT` is a good example for this: if used with a single argument, the argument should be numeric. When `LIMIT` is given a string value, executing the query will fail. You may want to detect this early and don't return an HTTP 500 (as this would signal attackers that they were successful breaking your application).

Another problem with `LIMIT` is that high `LIMIT` values are likely more expensive than low ones, and you may want to disallow using `LIMIT` values exceeding a certain threshold.

Here's what you could do in such cases:

```
var query = "FOR doc IN collection LIMIT @count RETURN doc";
```

```
/* some default value for limit */
var params = { count: 100 };

if (useLimit) {
  var count = req.params("count");

  /* abort if value does not look like an integer */
  if (! preg_match(/^d+$/, count)) {
    throw "invalid count value!";
  }

  /* actually turn it into an integer */
  params.count = parseInt(count, 10); // turn into numeric value
}

if (params.count < 1 || params.count > 1000) {
  /* value is outside of accepted thresholds */
  throw "invalid count value!";
}

db._query(query, params).toArray();
```

This is a bit more complex, but that's a price you're likely willing to pay for a bit of extra safety. In reality you may want to use a framework for validation (such as joi which comes bundled with ArangoDB) instead of writing your own checks all over the place.

Bind parameter types

There are two types of bind parameters in AQL:

- bind parameters for values: those are prefixed with a single `@` in AQL queries, and are specified without the prefix when they get their value assigned. These bind parameters can contain any valid JSON value.

  Examples: `@what`, `@searchValue`

- bind parameters for collections: these are prefixed with `@@` in AQL queries, and are replaced with the name of a collection. When the bind parameter value is assigned, the parameter itself must be specified with a single `@` prefix. Only string values are allowed for this type of bind parameters.

  Examples: `@@collection`

The latter type of bind parameter is probably not used as often, and it should not be used together with user input. Otherwise users may freely determine on which collection your AQL queries will operate (note: this may be a valid use case, but normally it is extremely undesired).

Authors: Jan Steemann

Tags: #injection #aql #security

Writing multi-line AQL queries

Problem

I want to write an AQL query that spans multiple lines in my JavaScript source code, but it does not work. How to do this?

Solution

AQL supports multi-line queries, and the AQL editor in ArangoDB's web interface supports them too.

When issued programmatically, multi-line queries can be a source of errors, at least in some languages. For example, JavaScript is notoriously bad at handling multi-line (JavaScript) statements, and until recently it had no support for multi-line strings.

In JavaScript, there are three ways of writing a multi-line AQL query in the source code:

- string concatenation
- ES6 template strings
- query builder

Which method works best depends on a few factors, but is often enough a simple matter of preference. Before deciding on any, please make sure to read the recipe for avoiding parameter injection too.

String concatenation

We want the query `FOR doc IN collection FILTER doc.value == @what RETURN doc` to become more legible in the source code.

Simply splitting the query string into three lines will leave us with a parse error in JavaScript:

```
/* will not work */
var query = "FOR doc IN collection
             FILTER doc.value == @what
             RETURN doc";
```

Instead, we could do this:

```
var query = "FOR doc IN collection " +
            "FILTER doc.value == @what " +
            "RETURN doc";
```

This is perfectly valid JavaScript, but it's error-prone. People have spent ages on finding subtle bugs in their queries because they missed a single whitespace character at the beginning or start of some line.

Please note that when assembling queries via string concatenation, you should still use bind parameters (as done above with `@what`) and not insert user input values into the query string without sanitation.

ES6 template strings

ES6 template strings are easier to get right and also look more elegant. They can be used inside ArangoDB since version 2.5. but some other platforms don't support them et. For example, they can't be used in IE and older node.js versions. So use them if your environment supports them and your code does not need to run on any non-ES6 environments.

Here's the query string declared via an ES6 template string (note that the string must be enclosed in backticks now):

```
var query = `FOR doc IN collection
             FILTER doc.value == @what
             RETURN doc`;
```

The whitespace in the template string-variant is much easier to get right than when doing the string concatenation.

There are a few things to note regarding template strings:

- ES6 template strings can be used to inject JavaScript values into the string dynamically. Substitutions start with the character sequence `${`. Care must be taken if this sequence itself is used inside the AQL query string (currently this would be invalid AQL, but this may change in future ArangoDB versions). Additionally, any values injected into the query string using parameter substitutions will not be escaped correctly automatically, so again special care must be taken when using this method to keep queries safe from parameter injection.
- a multi-line template string will actually contain newline characters. This is not necessarily the case when doing string concatenation. In the string concatenation example, we used three lines of source code to create a single-line query string. We could have inserted newlines into the query string there too, but we didn't. Just to point out that the two variants will not create bytewise-identical query strings.

Please note that when using ES6 template strings for your queries, you should still use bind parameters (as done above with `@what`) and not insert user input values into the query string without sanitation.

There is a convenience function `aql` which can be used to safely and easily build an AQL query with substitutions from arbitrary JavaScript values and expressions. It can be invoked like this:

```
const aql = require("@arangodb").aql; // not needed in arangosh

var what = "some input value";
var query = aql`FOR doc IN collection
                    FILTER doc.value == ${what}
                    RETURN doc`;
```

The template string variant that uses `aql` is both convenient and safe. Internally, it will turn the substituted values into bind parameters. The query string and the bind parameter values will be returned separately, so the result of `query` above will be something like:

```
{
  "query" : "FOR doc IN collection FILTER doc.value == @value0 RETURN doc",
  "bindVars" : {
    "value0" : "some input value"
  }
}
```

Query builder

ArangoDB comes bundled with a query builder named aqb. That query builder can be used to programmatically construct AQL queries, without having to write query strings at all.

Here's an example of its usage:

```
var qb = require("aqb");

var jobs = db._createStatement({
  query: (
    qb.for('job').in('_jobs')
    .filter(
      qb('pending').eq('job.status')
      .and(qb.ref('@queue').eq('job.queue'))
      .and(qb.ref('@now').gte('job.delayUntil'))
    )
    .sort('job.delayUntil', 'ASC')
    .limit('@max')
    .return('job')
  ),
  bindVars: {
    queue: queue._key,
    now: Date.now(),
    max: queue.maxWorkers - numBusy
  }
}).execute().toArray();
```

As can be seen, aqb provides a fluent API that allows chaining function calls for creating the individual query operations. This has a few advantages:

- flexibility: there is no query string in the source code, so the code can be formatted as desired without having to bother about strings
- validation: the query can be validated syntactically by aqb before being actually executed by the server. Testing of queries also becomes easier. Additionally, some IDEs may provide auto-completion to some extend and thus aid development
- security: built-in separation of query operations (e.g. `FOR`, `FILTER`, `SORT`, `LIMIT`) and dynamic values (e.g. user input values)

aqb can be used inside ArangoDB and from node.js and even from within browsers.

Authors: Jan Steemann

Tags: #aql #aqb #es6

Migrating GRAPH_* Functions from 2.8 or earlier to 3.0

Problem

With the release of 3.0 all GRAPH functions have been dropped from AQL in favor of a more native integration of graph features into the query language. I have used the old graph functions and want to upgrade to 3.0.

Graph functions covered in this recipe:

- GRAPH_COMMON_NEIGHBORS
- GRAPH_COMMON_PROPERTIES
- GRAPH_DISTANCE_TO
- GRAPH_EDGES
- GRAPH_NEIGHBORS
- GRAPH_TRAVERSAL
- GRAPH_TRAVERSAL_TREE
- GRAPH_SHORTEST_PATH
- GRAPH_PATHS
- GRAPH_VERTICES

Solution 1: Quick and Dirty (not recommended)

When to use this solution

I am not willing to invest a lot if time into the upgrade process and i am willing to surrender some performance in favor of less effort. Some constellations may not work with this solution due to the nature of user-defined functions. Especially check for AQL queries that do both modifications and `GRAPH_*` functions.

Registering user-defined functions

This step has to be executed once on ArangoDB for every database we are using.

We connect to `arangodb` with `arangosh` to issue the following commands two:

```
var graphs = require("@arangodb/general-graph");
graphs._registerCompatibilityFunctions();
```

These have registered all old `GRAPH_*` functions as user-defined functions again, with the prefix `arangodb::` .

Modify the application code

Next we have to go through our application code and replace all calls to `GRAPH_*` by `arangodb::GRAPH_*` . Now run a testrun of our application and check if it worked. If it worked we are ready to go.

Important Information

The user defined functions will call translated subqueries (as described in Solution 2). The optimizer does not know anything about these subqueries beforehand and cannot optimize the whole plan. Also there might be read/write constellations that are forbidden in user-defined functions, therefore a "really" translated query may work while the user-defined function work around may be rejected.

Solution 2: Translating the queries (recommended)

When to use this solution

I am willing to invest some time on my queries in order to get maximum performance, full query optimization and a better control of my queries. No forcing into the old layout any more.

Before you start

If you are using `vertexExamples` which are not only `_id` strings do not skip the GRAPH_VERTICES section, because it will describe how to translate them to AQL. All graph functions using a vertexExample are identical to executing a GRAPH_VERTICES before and using it's result as start point. Example with NEIGHBORS:

```
FOR res IN GRAPH_NEIGHBORS(@graph, @myExample) RETURN res
```

Is identical to:

```
FOR start GRAPH_VERTICES(@graph, @myExample)
  FOR res IN GRAPH_NEIGHBORS(@graph, start) RETURN res
```

All non GRAPH_VERTICES functions will only explain the transformation for a single input document's `_id`.

Options used everywhere

Option edgeCollectionRestriction

In order to use edge Collection restriction we just use the feature that the traverser can walk over a list of edge collections directly. So the edgeCollectionRestrictions just form this list (exampleGraphEdges):

```
// OLD
[..] FOR e IN GRAPH_EDGES(@graphName, @startId, {edgeCollectionRestriction: [edges1, edges2]}) RETURN e

// NEW
[..] FOR v, e IN ANY @startId edges1, edges2 RETURN DISTINCT e._id
```

Note: The `@graphName` bindParameter is not used anymore and probably has to be removed from the query.

Option includeData

If we use the option includeData we simply return the object directly instead of only the _id

Example GRAPH_EDGES:

```
// OLD
[..] FOR e IN GRAPH_EDGES(@graphName, @startId, {includeData: true}) RETURN e

// NEW
[..] FOR v, e IN ANY @startId GRAPH @graphName RETURN DISTINCT e
```

Option direction

The direction has to be placed before the start id. Note here: The direction has to be placed as Word it cannot be handed in via a bindParameter anymore:

```
// OLD
[..] FOR e IN GRAPH_EDGES(@graphName, @startId, {direction: 'inbound'}) RETURN e

// NEW
[..] FOR v, e IN INBOUND @startId GRAPH @graphName RETURN DISTINCT e._id
```

Options minDepth, maxDepth

If we use the options minDepth and maxDepth (both default 1 if not set) we can simply put them in front of the direction part in the Traversal statement.

Example GRAPH_EDGES:

```
// OLD
[..] FOR e IN GRAPH_EDGES(@graphName, @startId, {minDepth: 2, maxDepth: 4}) RETURN e

// NEW
[..] FOR v, e IN 2..4 ANY @startId GRAPH @graphName RETURN DISTINCT e._id
```

Option maxIteration

The option `maxIterations` is removed without replacement. Your queries are now bound by main memory not by an arbitrary number of iterations.

GRAPH_VERTICES

First we have to branch on the example. There we have three possibilities:

1. The example is an `_id` string.
2. The example is `null` or `{}` .
3. The example is a non empty object or an array.

Example is '_id' string

This is the easiest replacement. In this case we simply replace the function with a call to `DOCUMENT` :

```
// OLD
[..] GRAPH_VERTICES(@graphName, @idString) [..]

// NEW
[..] DOCUMENT(@idString) [..]
```

NOTE: The `@graphName` is not required anymore, we may have to adjust bindParameters.

The AQL graph features can work with an id directly, no need to call `DOCUMENT` before if we just need this to find a starting point.

Example is `null` or the empty object

This case means we use all documents from the graph. Here we first have to now the vertex collections of the graph.

1. If we only have one collection (say `vertices`) we can replace it with a simple iteration over this collection:

```
// OLD
[..] FOR v IN GRAPH_VERTICES(@graphName, {}) [..]

// NEW
[..] FOR v IN vertices [..]
`
```

NOTE: The `@graphName` is not required anymore, we may have to adjust bindParameters.

1. We have more than one collection. This is the unfortunate case for a general replacement. So in the general replacement we assume we do not want to exclude any of the collections in the graph. Than we unfortunately have to form a `UNION` over all these collections. Say our graph has the vertex collections `vertices1` , `vertices2` , `vertices3` we create a sub-query for a single collection for each of them and wrap them in a call to `UNION` .

```
// OLD
[..] FOR v IN GRAPH_VERTICES(@graphName, {}) [..]

// NEW
[..]
FOR v IN UNION( // We start a UNION
  (FOR v IN vertices1 RETURN v), // For each vertex collection
  (FOR v IN vertices2 RETURN v), // we create the same subquery
  (FOR v IN vertices3 RETURN v)
```

```
) // Finish with the UNION
[..]
`
```

NOTE: If you have any more domain knowledge of your graph apply it at this point to identify which collections are actually relevant as this `UNION` is a rather expensive operation.

If we use the option `vertexCollectionRestriction` in the original query. The `UNION` has to be formed by the collections in this restriction instead of ALL collections.

Example is a non-empty object

First we follow the instructions for the empty object above. In this section we will just focus on a single collection `vertices`, the UNION for multiple collections is again wrapped around a subquery for each of these collections built in the following way.

Now we have to transform the example into an AQL `FILTER` statement. Therefore we take all top-level attributes of the example and do an equal comparison with their values. All of these comparisons are joined with an `AND` because the all have to be fulfilled.

Example:

```
// OLD
[..] FOR v IN GRAPH_VERTICES(@graphName, {foo: 'bar', the: {answer: 42}}}) [..]

// NEW
[..] FOR v IN vertices
  FILTER v.foo == 'bar' // foo: bar
  AND    v.the == {answer: 42} //the: {answer: 42}
[..]
```

Example is an array

The idea transformation is almost identical to a single non-empty object. For each element in the array we create the filter conditions and than we `OR`-combine them (mind the brackets):

```
// OLD
[..] FOR v IN GRAPH_VERTICES(@graphName, [{foo: 'bar', the: {answer: 42}}, {foo: 'baz'}])) [..]

// NEW
[..] FOR v IN vertices
  FILTER (v.foo == 'bar' // foo: bar
    AND   v.the == {answer: 42}) //the: {answer: 42}
  OR (v.foo == 'baz')
[..]
```

GRAPH_EDGES

The GRAPH_EDGES can be simply replaced by a call to the AQL traversal.

No options

The default options did use a direction `ANY` and returned a distinct result of the edges. Also it did just return the edges `_id` value.

```
// OLD
[..] FOR e IN GRAPH_EDGES(@graphName, @startId) RETURN e

// NEW
[..] FOR v, e IN ANY @startId GRAPH @graphName RETURN DISTINCT e._id
```

Option edgeExamples.

See `GRAPH_VERTICES` on how to transform examples to AQL FILTER. Apply the filter on the edge variable `e`.

GRAPH_NEIGHBORS

The GRAPH_NEIGHBORS is a breadth-first-search on the graph with a global unique check for vertices. So we can replace it by a an AQL traversal with these options.

No options

The default options did use a direction `ANY` and returned a distinct result of the neighbors. Also it did just return the neighbors `_id` value.

```
// OLD
[..] FOR n IN GRAPH_NEIGHBORS(@graphName, @startId) RETURN n

// NEW
[..] FOR n IN ANY @startId GRAPH @graphName OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN n
```

Option neighborExamples

See `GRAPH_VERTICES` on how to transform examples to AQL FILTER. Apply the filter on the neighbor variable `n`.

Option edgeExamples

See `GRAPH_VERTICES` on how to transform examples to AQL FILTER. Apply the filter on the edge variable `e`.

However this is a bit more complicated as it interferes with the global uniqueness check. For edgeExamples it is sufficent when any edge pointing to the neighbor matches the filter. Using `{uniqueVertices: 'global'}` first picks any edge randomly. Than it checks against this edge only. If we know there are no vertex pairs with multiple edges between them we can use the simple variant which is save:

```
// OLD
[..] FOR n IN GRAPH_NEIGHBORS(@graphName, @startId, {edgeExample: {label: 'friend'}}) RETURN e

// NEW
[..] FOR n, e IN ANY @startId GRAPH @graphName OPTIONS {bfs: true, uniqueVertices: 'global'} FILTER e.label == 'friend' RETURN
n._id
```

If there may be multiple edges between the same pair of vertices we have to make the distinct check ourselfes and cannot rely on the traverser doing it correctly for us:

```
// OLD
[..] FOR n IN GRAPH_NEIGHBORS(@graphName, @startId, {edgeExample: {label: 'friend'}}) RETURN e

// NEW
[..] FOR n, e IN ANY @startId GRAPH @graphName OPTIONS {bfs: true} FILTER e.label == 'friend' RETURN DISTINCT n._id
```

Option vertexCollectionRestriction

If we use the vertexCollectionRestriction we have to postFilter the neighbors based on their collection. Therefore we can make use of the function `IS_SAME_COLLECTION`:

```
// OLD
[..] FOR n IN GRAPH_NEIGHBORS(@graphName, @startId, {vertexCollectionRestriction: ['vertices1', 'vertices2']}) RETURN e

// NEW
[..] FOR n IN ANY @startId GRAPH @graphName OPTIONS {bfs: true, uniqueVertices: true} FILTER IS_SAME_COLLECTION(vertices1, n) O
R IS_SAME_COLLECTION(vertices2, n) RETURN DISTINCT n._id
```

GRAPH_COMMON_NEIGHBORS

`GRAPH_COMMON_NEIGHBORS` is defined as two `GRAPH_NEIGHBORS` queries and than forming the `INTERSECTION` of both queries. How to translate the options please refer to `GRAPH_NEIGHBORS`. Finally we have to build the old result format `{left, right, neighbors}`. If you just need parts of the result you can adapt this query to your specific needs.

```
// OLD
```

```
FOR v IN GRAPH_COMMON_NEIGHBORS(@graphName, 'vertices/1' , 'vertices/2',  {direction : 'any'}) RETURN v

// NEW
LET n1 = ( // Neighbors for vertex1Example
  FOR n IN ANY 'vertices/1' GRAPH 'graph' OPTIONS {bfs: true, uniqueVertices: "global"} RETURN n._id
  )
LET n2 = ( // Neighbors for vertex2Example
  FOR n IN ANY 'vertices/2' GRAPH 'graph' OPTIONS {bfs: true, uniqueVertices: "global"} RETURN n._id
  )
LET common = INTERSECTION(n1, n2) // Get the intersection
RETURN { // Produce the original result
  left: 'vertices/1',
  right: 'vertices/2,
  neighbors: common
}
```

NOTE: If you are using examples instead of `_ids` you have to add a filter to make sure that the left is not equal to the right start vertex. To give you an example with a single vertex collection `vertices`, the replacement would look like this:

```
// OLD
FOR v IN GRAPH_COMMON_NEIGHBORS(@graphName, {name: "Alice"}, {name: "Bob"}) RETURN v

// NEW
FOR left IN vertices
  FILTER left.name == "Alice"
  LET n1 = (FOR n IN ANY left GRAPH 'graph' OPTIONS {bfs: true, uniqueVertices: "global"} RETURN n._id)
  FOR right IN vertices
    FILTER right.name == "Bob"
    FILTER right != left // Make sure left is not identical to right
    LET n2 = (FOR n IN ANY right GRAPH 'graph' OPTIONS {bfs: true, uniqueVertices: "global"} RETURN n._id)
    LET neighbors = INTERSECTION(n1, n2)
    FILTER LENGTH(neighbors) > 0 // Only pairs with shared neighbors should be returned
    RETURN {left: left._id, right: right._id, neighbors: neighbors}
```

GRAPH_PATHS

This function computes all paths of the entire graph (with a given minDepth and maxDepth) as you can imagine this feature is extremely expensive and should never be used. However paths can again be replaced by AQL traversal. Assume we only have one vertex collection `vertices` again.

No options

By default paths of length 0 to 10 are returned. And circles are not followed.

```
// OLD
RETURN GRAPH_PATHS('graph')

// NEW
FOR start IN vertices
FOR v, e, p IN 0..10 OUTBOUND start GRAPH 'graph' RETURN {source: start, destination: v, edges: p.edges, vertices: p.vertices}
```

followCycles

If this option is set we have to modify the options of the traversal by modifying the `uniqueEdges` property:

```
// OLD
RETURN GRAPH_PATHS('graph', {followCycles: true})

// NEW
FOR start IN vertices
FOR v, e, p IN 0..10 OUTBOUND start GRAPH 'graph' OPTIONS {uniqueEdges: 'none'} RETURN {source: start, destination: v, edges: p
.edges, vertices: p.vertices}
```

GRAPH_COMMON_PROPERTIES

This feature involves several full-collection scans and therefore is extremely expensive. If you really need it you can transform it with the help of `ATTRIBUTES`, `KEEP` and `ZIP`.

Start with single _id

```
// OLD
RETURN GRAPH_COMMON_PROPERTIES('graph', "vertices/1", "vertices/2")

// NEW
LET left = DOCUMENT("vertices/1") // get one document
LET right = DOCUMENT("vertices/2") // get the other one
LET shared = (FOR a IN ATTRIBUTES(left) // find all shared attributes
               FILTER left[a] == right[a]
                 OR a == '_id' // always include _id
                 RETURN a)
FILTER LENGTH(shared) > 1 // Return them only if they share an attribute
RETURN ZIP([left._id], [KEEP(right, shared)]) // Build the result
```

Start with vertexExamples

Again we assume we only have a single collection `vertices`. We have to transform the examples into filters. Iterate over vertices to find all left documents. For each left document iterate over vertices again to find matching right documents. And return the shared attributes as above:

```
// OLD
RETURN GRAPH_COMMON_PROPERTIES('graph', {answer: 42}, {foo: "bar"})

// NEW
FOR left IN vertices
  FILTER left.answer == 42
  LET commons = (
    FOR right IN vertices
      FILTER right.foo == "bar"
      FILTER left != right
      LET shared = (FOR a IN ATTRIBUTES(left)
                      FILTER left[a] == right[a]
                      OR a == '_id'
                        RETURN a)
      FILTER LENGTH(shared) > 1
      RETURN KEEP(right, shared))
  FILTER LENGTH(commons) > 0
  RETURN ZIP([left._id], [commons])
```

GRAPH_SHORTEST_PATH

A shortest path computation is now done via the new SHORTEST_PATH AQL statement.

No options

```
// OLD
FOR p IN GRAPH_SHORTEST_PATH(@graphName, @startId, @targetId, {direction : 'outbound'}) RETURN p

// NEW
LET p = ( // Run one shortest Path
  FOR v, e IN OUTBOUND SHORTEST_PATH @startId TO @targetId GRAPH @graphName
  // We return objects with vertex, edge and weight for each vertex on the path
  RETURN {vertex: v, edge: e, weight: (IS_NULL(e) ? 0 : 1)}
)
FILTER LENGTH(p) > 0 // We only want shortest paths that actually exist
RETURN { // We rebuild the old format
  vertices: p[*].vertex,
  edges: p[* FILTER CURRENT.e != null].edge,
  distance: SUM(p[*].weight)
}
```

Options weight and defaultWeight

The new AQL SHORTEST_PATH offers the options `weightAttribute` and `defaultWeight`.

```
// OLD
FOR p IN GRAPH_SHORTEST_PATH(@graphName, @startId, @targetId, {direction : 'outbound', weight: "weight", defaultWeight: 80}) RE
TURN p

// NEW
LET p = ( // Run one shortest Path
  FOR v, e IN OUTBOUND SHORTEST_PATH @startId TO @targetId GRAPH @graphName
  // We return objects with vertex, edge and weight for each vertex on the path
  RETURN {vertex: v, edge: e, weight: (IS_NULL(e) ? 0 : (IS_NUMBER(e.weight) ? e.weight : 80))}
)
FILTER LENGTH(p) > 0 // We only want shortest paths that actually exist
RETURN { // We rebuild the old format
  vertices: p[*].vertex,
  edges: p[* FILTER CURRENT.e != null].edge,
  distance: SUM(p[*].weight) // We have to recompute the distance if we need it
}
```

GRAPH_DISTANCE_TO

Graph distance to only differs by the result format from `GRAPH_SHORTEST_PATH`. So we follow the transformation for `GRAPH_SHORTEST_PATH`, remove some unnecessary parts, and change the return format

```
// OLD
FOR p IN GRAPH_DISTANCE_TO(@graphName, @startId, @targetId, {direction : 'outbound'}) RETURN p

// NEW
LET p = ( // Run one shortest Path
  FOR v, e IN OUTBOUND SHORTEST_PATH @startId TO @targetId GRAPH @graphName
  // DIFFERENCE we only return the weight for each edge on the path
  RETURN IS_NULL(e) ? 0 : 1}
)
FILTER LENGTH(p) > 0 // We only want shortest paths that actually exist
RETURN { // We rebuild the old format
  startVertex: @startId,
  vertex: @targetId,
  distance: SUM(p[*].weight)
}
```

GRAPH_TRAVERSAL and GRAPH_TRAVERSAL_TREE

These have been removed and should be replaced by the native AQL traversal. There are many potential solutions using the new syntax, but they largely depend on what exactly you are trying to achieve and would go beyond the scope of this cookbook. Here is one example how to do the transition, using the world graph as data:

In 2.8, it was possible to use `GRAPH_TRAVERSAL()` together with a custom visitor function to find leaf nodes in a graph. Leaf nodes are vertices that have inbound edges, but no outbound edges. The visitor function code looked like this:

```
var aqlfunctions = require("org/arangodb/aql/functions");

aqlfunctions.register("myfunctions::leafNodeVisitor", function (config, result, vertex, path, connected) {
  if (connected && connected.length === 0) {
    return vertex.name + " (" + vertex.type + ")";
  }
});
```

And the AQL query to make use of it:

```
LET params = {
  order: "preorder-expander",
  visitor: "myfunctions::leafNodeVisitor",
  visitorReturnsResults: true
}
FOR result IN GRAPH_TRAVERSAL("worldCountry", "worldVertices/world", "inbound", params)
  RETURN result
```

To traverse the graph starting at vertex `worldVertices/world` using native AQL traversal and a named graph, we can simply do:

```
FOR v IN 0..10 INBOUND "worldVertices/world" GRAPH "worldCountry"
  RETURN v
```

It will give us all vertex documents including the start vertex (because the minimum depth is set to *0*). The maximum depth is set to *10*, which is enough to follow all edges and reach the leaf nodes in this graph.

The query can be modified to return a formatted path from first to last node:

```
FOR v, e, p IN 0..10 INBOUND "worldVertices/world" GRAPH "worldCountry"
  RETURN CONCAT_SEPARATOR(" -> ", p.vertices[*].name)
```

The result looks like this (shortened):

```
[
  "World",
  "World -> Africa",
  "World -> Africa -> Cote d'Ivoire",
  "World -> Africa -> Cote d'Ivoire -> Yamoussoukro",
  "World -> Africa -> Angola",
  "World -> Africa -> Angola -> Luanda",
  "World -> Africa -> Chad",
  "World -> Africa -> Chad -> N'Djamena",
  ...
]
```

As we can see, all possible paths of varying lengths are returned. We are not really interested in them, but we still have to do the traversal to go from *World* all the way to the leaf nodes (e.g. *Yamoussoukro*). To determine if a vertex is really the last on the path in the sense of being a leaf node, we can use another traversal of depth 1 to check if there is at least one outgoing edge - which means the vertex is not a leaf node, otherwise it is:

```
FOR v IN 0..10 INBOUND "worldVertices/world" GRAPH "worldCountry"
  FILTER LENGTH(FOR vv IN INBOUND v GRAPH "worldCountry" LIMIT 1 RETURN 1) == 0
  RETURN CONCAT(v.name, " (", v.type, ")")
```

Using the current vertex `v` as starting point, the second traversal is performed. It can return early after one edge was followed (`LIMIT 1`), because we don't need to know the exact count and it is faster this way. We also don't need the actual vertex, so we can just `RETURN 1` as dummy value as an optimization. The traversal (which is a sub-query) will return an empty array in case of a leaf node, and `[ 1 ]` otherwise. Since we only want the leaf nodes, we `FILTER` out all non-empty arrays and what is left are the leaf nodes only. The attributes `name` and `type` are formatted the way they were like in the original JavaScript code, but now with AQL. The final result is a list of all capitals:

```
[
  "Yamoussoukro (capital)",
  "Luanda (capital)",
  "N'Djamena (capital)",
  "Algiers (capital)",
  "Yaounde (capital)",
  "Ouagadougou (capital)",
  "Gaborone (capital)",
  "Asmara (capital)",
  "Cairo (capital)",
  ...
]
```

There is no direct substitute for the `GRAPH_TRAVERSAL_TREE()` function. The advantage of this function was that its (possibly highly nested) result data structure inherently represented the "longest" possible paths only. With native AQL traversal, all paths from minimum to maximum traversal depth are returned, including the "short" paths as well:

```
FOR v, e, p IN 1..2 INBOUND "worldVertices/continent-north-america" GRAPH "worldCountry"
  RETURN CONCAT_SEPARATOR(" <- ", p.vertices[*]._key)
```

```
[
  "continent-north-america <- country-antigua-and-barbuda",
  "continent-north-america <- country-antigua-and-barbuda <- capital-saint-john-s",
  "continent-north-america <- country-barbados",
  "continent-north-america <- country-barbados <- capital-bridgetown",
  "continent-north-america <- country-canada",
  "continent-north-america <- country-canada <- capital-ottawa",
  "continent-north-america <- country-bahamas",
  "continent-north-america <- country-bahamas <- capital-nassau"
]
```

A second traversal with depth = 1 can be used to check if we reached a leaf node (no more incoming edges). Based on this information, the "short" paths can be filtered out. Note that a second condition is required: it is possible that the last node in a traversal is not a leaf node if the maximum traversal depth is exceeded. Thus, we need to also let paths through, which contain as many edges as hops we do in the traversal (here: 2).

```
FOR v, e, p IN 1..2 INBOUND "worldVertices/continent-north-america" GRAPH "worldCountry"
  LET other = (
    FOR vv, ee IN INBOUND v GRAPH "worldCountry"
      //FILTER ee != e // needed if traversing edges in ANY direction
      LIMIT 1
      RETURN 1
  )
  FILTER LENGTH(other) == 0 || LENGTH(p.edges) == 2
  RETURN CONCAT_SEPARATOR(" <- ", p.vertices[*]._key)
```

```
[
  "continent-north-america <- country-antigua-and-barbuda <- capital-saint-john-s",
  "continent-north-america <- country-barbados <- capital-bridgetown",
  "continent-north-america <- country-canada <- capital-ottawa",
  "continent-north-america <- country-bahamas <- capital-nassau"
]
```

The full paths can be returned, but it is not in a tree-like structure as with `GRAPH_TRAVERSAL_TREE()`. Such a data structure can be constructed on client-side if really needed.

```
FOR v, e, p IN 1..2 INBOUND "worldVertices/continent-north-america" GRAPH "worldCountry"
  LET other = (FOR vv, ee IN INBOUND v GRAPH "worldCountry" LIMIT 1 RETURN 1)
  FILTER LENGTH(other) == 0 || LENGTH(p.edges) == 2
  RETURN p
```

Path data (shortened):

```
[
  {
    "edges": [
      {
        "_id": "worldEdges/57585025",
        "_from": "worldVertices/country-antigua-and-barbuda",
        "_to": "worldVertices/continent-north-america",
        "type": "is-in"
      },
      {
        "_id": "worldEdges/57585231",
        "_from": "worldVertices/capital-saint-john-s",
        "_to": "worldVertices/country-antigua-and-barbuda",
        "type": "is-in"
      }
    ],
    "vertices": [
      {
        "_id": "worldVertices/continent-north-america",
        "name": "North America",
        "type": "continent"
      },
      {
        "_id": "worldVertices/country-antigua-and-barbuda",
        "code": "ATG",
```

```
        "name": "Antigua and Barbuda",
        "type": "country"
      },
      {
        "_id": "worldVertices/capital-saint-john-s",
        "name": "Saint John's",
        "type": "capital"
      }
    ]
  },
  {
    ...
  }
]
```

The first and second vertex of the nth path are connected by the first edge (`p[n].vertices[0]` `p[n].edges[0]` → `p[n].vertices[1]`) and so on. This structure might actually be more convenient to process compared to a tree-like structure. Note that the edge documents are also included, in constrast to the removed graph traversal function.

Contact us via our social channels if you need further help.

Author: Michael Hackstein

Tags: #howto #aql #migration

Migrating anonymous graph Functions from 2.8 or earlier to 3.0

Problem

With the release of 3.0 all GRAPH functions have been dropped from AQL in favor of a more native integration of graph features into the query language. I have used the old graph functions and want to upgrade to 3.0.

Graph functions covered in this recipe:

- EDGES
- NEIGHBORS
- PATHS
- TRAVERSAL
- TRAVERSAL_TREE

Solution

EDGES

The EDGES can be simply replaced by a call to the AQL traversal.

No options

The syntax is slightly different but mapping should be simple:

```
// OLD
[..] FOR e IN EDGES(@@edgeCollection, @startId, 'outbound') RETURN e

// NEW
[..] FOR v, e IN OUTBOUND @startId @@edgeCollection RETURN e
```

Using EdgeExamples

Examples have to be transformed into AQL filter statements. How to do this please read the GRAPH_VERTICES section in Migrating GRAPH_* Functions from 2.8 or earlier to 3.0. Apply these filters on the edge variable `e`.

Option incluceVertices

In order to include the vertices you just use the vertex variable v as well:

```
// OLD
[..] FOR e IN EDGES(@@edgeCollection, @startId, 'outbound', null, {includeVertices: true}) RETURN e

// NEW
[..] FOR v, e IN OUTBOUND @startId @@edgeCollection RETURN {edge: e, vertex: v}
```

NOTE: The direction cannot be given as a bindParameter any more it has to be hard-coded in the query.

NEIGHBORS

The NEIGHBORS is a breadth-first-search on the graph with a global unique check for vertices. So we can replace it by a an AQL traversal with these options. Due to syntax changes the vertex collection of the start vertex is no longer mandatory to be given. You may have to adjust bindParameteres for this query.

No options

The default options did just return the neighbors `_id` value.

```
// OLD
[..] FOR n IN NEIGHBORS(@@vertexCollection, @@edgeCollection, @startId, 'outbound') RETURN n

// NEW
[..] FOR n IN OUTBOUND @startId @@edgeCollection OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN n._id
```

NOTE: The direction cannot be given as a bindParameter any more it has to be hard-coded in the query.

Using edgeExamples

Examples have to be transformed into AQL filter statements. How to do this please read the GRAPH_VERTICES section in Migrating GRAPH_* Functions from 2.8 or earlier to 3.0. Apply these filters on the edge variable `e` which is the second return variable of the traversal statement.

However this is a bit more complicated as it interferes with the global uniqueness check. For edgeExamples it is sufficent when any edge pointing to the neighbor matches the filter. Using `{uniqueVertices: 'global'}` first picks any edge randomly. Than it checks against this edge only. If we know there are no vertex pairs with multiple edges between them we can use the simple variant which is save:

```
// OLD
[..] FOR n IN NEIGHBORS(@@vertexCollection, @@edgeCollection, @startId, 'outbound', {label: 'friend'}) RETURN n

// NEW
[..] FOR n, e IN OUTBOUND @startId @@edgeCollection OPTIONS {bfs: true, uniqueVertices: 'global'}
FILTER e.label == 'friend'
RETURN n._id
```

If there may be multiple edges between the same pair of vertices we have to make the distinct check ourselfes and cannot rely on the traverser doing it correctly for us:

```
// OLD
[..] FOR n IN NEIGHBORS(@@vertexCollection, @@edgeCollection, @startId, 'outbound', {label: 'friend'}) RETURN n

// NEW
[..] FOR n, e IN OUTBOUND @startId @@edgeCollection OPTIONS {bfs: true}
FILTER e.label == 'friend'
RETURN DISTINCT n._id
```

Option includeData

If you want to include the data simply return the complete document instead of only the `_id` value.

```
// OLD
[..] FOR n IN NEIGHBORS(@@vertexCollection, @@edgeCollection, @startId, 'outbound', null, {includeData: true}) RETURN n

// NEW
[..] FOR n, e IN OUTBOUND @startId @@edgeCollection OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN n
```

PATHS

This function computes all paths of the entire edge collection (with a given minDepth and maxDepth) as you can imagine this feature is extremely expensive and should never be used. However paths can again be replaced by AQL traversal.

No options

By default paths of length 0 to 10 are returned. And circles are not followed.

```
// OLD
RETURN PATHS(@@vertexCollection, @@edgeCollection, "outbound")

// NEW
FOR start IN @@vertexCollection
```

```
FOR v, e, p IN 0..10 OUTBOUND start @@edgeCollection RETURN {source: start, destination: v, edges: p.edges, vertices: p.vertice
s}
```

followCycles

If this option is set we have to modify the options of the traversal by modifying the `uniqueEdges` property:

```
// OLD
RETURN PATHS(@@vertexCollection, @@edgeCollection, "outbound", {followCycles: true})

// NEW
FOR start IN @@vertexCollection
FOR v, e, p IN 0..10 OUTBOUND start @@edgeCollection OPTIONS {uniqueEdges: 'none'} RETURN {source: start, destination: v, edges
: p.edges, vertices: p.vertices}
```

minDepth and maxDepth

If this option is set we have to give these parameters directly before the direction.

```
// OLD
RETURN PATHS(@@vertexCollection, @@edgeCollection, "outbound", {minDepth: 2, maxDepth: 5})

// NEW
FOR start IN @@vertexCollection
FOR v, e, p IN 2..5 OUTBOUND start @@edgeCollection
RETURN {source: start, destination: v, edges: p.edges, vertices: p.vertices}
```

TRAVERSAL and TRAVERSAL_TREE

These have been removed and should be replaced by the native AQL traversal. There are many potential solutions using the new syntax, but they largely depend on what exactly you are trying to achieve and would go beyond the scope of this cookbook. Here is one example how to do the transition, using the world graph as data:

In 2.8, it was possible to use `TRAVERSAL()` together with a custom visitor function to find leaf nodes in a graph. Leaf nodes are vertices that have inbound edges, but no outbound edges. The visitor function code looked like this:

```
var aqlfunctions = require("org/arangodb/aql/functions");

aqlfunctions.register("myfunctions::leafNodeVisitor", function (config, result, vertex, path, connected) {
  if (connected && connected.length === 0) {
    return vertex.name + " (" + vertex.type + ")";
  }
});
```

And the AQL query to make use of it:

```
LET params = {
  order: "preorder-expander",
  visitor: "myfunctions::leafNodeVisitor",
  visitorReturnsResults: true
}
FOR result IN TRAVERSAL(worldVertices, worldEdges, "worldVertices/world", "inbound", params)
  RETURN result
```

To traverse the graph starting at vertex `worldVertices/world` using native AQL traversal and an anonymous graph, we can simply do:

```
FOR v IN 0..10 INBOUND "worldVertices/world" worldEdges
  RETURN v
```

It will give us all vertex documents including the start vertex (because the minimum depth is set to *0*). The maximum depth is set to *10*, which is enough to follow all edges and reach the leaf nodes in this graph.

The query can be modified to return a formatted path from first to last node:

```
FOR v, e, p IN 0..10 INBOUND "worldVertices/world" e
  RETURN CONCAT_SEPARATOR(" -> ", p.vertices[*].name)
```

The result looks like this (shortened):

```
[
  "World",
  "World -> Africa",
  "World -> Africa -> Cote d'Ivoire",
  "World -> Africa -> Cote d'Ivoire -> Yamoussoukro",
  "World -> Africa -> Angola",
  "World -> Africa -> Angola -> Luanda",
  "World -> Africa -> Chad",
  "World -> Africa -> Chad -> N'Djamena",
  ...
]
```

As we can see, all possible paths of varying lengths are returned. We are not really interested in them, but we still have to do the traversal to go from *World* all the way to the leaf nodes (e.g. *Yamoussoukro*). To determine if a vertex is really the last on the path in the sense of being a leaf node, we can use another traversal of depth 1 to check if there is at least one outgoing edge - which means the vertex is not a leaf node, otherwise it is:

```
FOR v IN 0..10 INBOUND "worldVertices/world" worldEdges
  FILTER LENGTH(FOR vv IN INBOUND v worldEdges LIMIT 1 RETURN 1) == 0
  RETURN CONCAT(v.name, " (", v.type, ")")
```

Using the current vertex `v` as starting point, the second traversal is performed. It can return early after one edge was followed (`LIMIT 1`), because we don't need to know the exact count and it is faster this way. We also don't need the actual vertex, so we can just `RETURN 1` as dummy value as an optimization. The traversal (which is a sub-query) will return an empty array in case of a leaf node, and `[ 1 ]` otherwise. Since we only want the leaf nodes, we `FILTER` out all non-empty arrays and what is left are the leaf nodes only. The attributes `name` and `type` are formatted the way they were like in the original JavaScript code, but now with AQL. The final result is a list of all capitals:

```
[
  "Yamoussoukro (capital)",
  "Luanda (capital)",
  "N'Djamena (capital)",
  "Algiers (capital)",
  "Yaounde (capital)",
  "Ouagadougou (capital)",
  "Gaborone (capital)",
  "Asmara (capital)",
  "Cairo (capital)",
  ...
]
```

There is no direct substitute for the `TRAVERSAL_TREE()` function. The advantage of this function was that its (possibly highly nested) result data structure inherently represented the "longest" possible paths only. With native AQL traversal, all paths from minimum to maximum traversal depth are returned, including the "short" paths as well:

```
FOR v, e, p IN 1..2 INBOUND "worldVertices/continent-north-america" worldEdges
  RETURN CONCAT_SEPARATOR(" <- ", p.vertices[*]._key)
```

```
[
  "continent-north-america <- country-antigua-and-barbuda",
  "continent-north-america <- country-antigua-and-barbuda <- capital-saint-john-s",
  "continent-north-america <- country-barbados",
  "continent-north-america <- country-barbados <- capital-bridgetown",
  "continent-north-america <- country-canada",
  "continent-north-america <- country-canada <- capital-ottawa",
  "continent-north-america <- country-bahamas",
  "continent-north-america <- country-bahamas <- capital-nassau"
]
```

A second traversal with depth = 1 can be used to check if we reached a leaf node (no more incoming edges). Based on this information, the "short" paths can be filtered out. Note that a second condition is required: it is possible that the last node in a traversal is not a leaf node if the maximum traversal depth is exceeded. Thus, we need to also let paths through, which contain as many edges as hops we do in the traversal (here: 2).

```
FOR v, e, p IN 1..2 INBOUND "worldVertices/continent-north-america" worldEdges
  LET other = (
    FOR vv, ee IN INBOUND v worldEdges
      //FILTER ee != e // needed if traversing edges in ANY direction
      LIMIT 1
      RETURN 1
  )
  FILTER LENGTH(other) == 0 || LENGTH(p.edges) == 2
  RETURN CONCAT_SEPARATOR(" <- ", p.vertices[*]._key)
```

```
[
  "continent-north-america <- country-antigua-and-barbuda <- capital-saint-john-s",
  "continent-north-america <- country-barbados <- capital-bridgetown",
  "continent-north-america <- country-canada <- capital-ottawa",
  "continent-north-america <- country-bahamas <- capital-nassau"
]
```

The full paths can be returned, but it is not in a tree-like structure as with `TRAVERSAL_TREE()`. Such a data structure can be constructed on client-side if really needed.

```
FOR v, e, p IN 1..2 INBOUND "worldVertices/continent-north-america" worldEdges
  LET other = (FOR vv, ee IN INBOUND v worldEdges LIMIT 1 RETURN 1)
  FILTER LENGTH(other) == 0 || LENGTH(p.edges) == 2
  RETURN p
```

Path data (shortened):

```
[
  {
    "edges": [
      {
        "_id": "worldEdges/57585025",
        "_from": "worldVertices/country-antigua-and-barbuda",
        "_to": "worldVertices/continent-north-america",
        "type": "is-in"
      },
      {
        "_id": "worldEdges/57585231",
        "_from": "worldVertices/capital-saint-john-s",
        "_to": "worldVertices/country-antigua-and-barbuda",
        "type": "is-in"
      }
    ],
    "vertices": [
      {
        "_id": "worldVertices/continent-north-america",
        "name": "North America",
        "type": "continent"
      },
      {
        "_id": "worldVertices/country-antigua-and-barbuda",
        "code": "ATG",
        "name": "Antigua and Barbuda",
        "type": "country"
      },
      {
        "_id": "worldVertices/capital-saint-john-s",
        "name": "Saint John's",
        "type": "capital"
      }
    ]
  },
  {
    ...
```

```
  }
]
```

The first and second vertex of the nth path are connected by the first edge (`p[n].vertices[0]` `p[n].edges[0]` → `p[n].vertices[1]`) and so on. This structure might actually be more convenient to process compared to a tree-like structure. Note that the edge documents are also included, in constrast to the removed graph traversal function.

Contact us via our social channels if you need further help.

Author: Michael Hackstein

Tags: #howto #aql #migration

Migrating GRAPH_* Measurements from 2.8 or earlier to 3.0

Problem

With the release of 3.0 all GRAPH functions have been dropped from AQL in favor of a more native integration of graph features into the query language. I have used the old graph functions and want to upgrade to 3.0.

Graph functions covered in this recipe:

- GRAPH_ABSOLUTE_BETWEENNESS
- GRAPH_ABSOLUTE_CLOSENESS
- GRAPH_ABSOLUTE_ECCENTRICITY
- GRAPH_BETWEENNESS
- GRAPH_CLOSENESS
- GRAPH_DIAMETER
- GRAPH_ECCENTRICITY
- GRAPH_RADIUS

Solution 1: User Defined Funtions

Registering user-defined functions

This step has to be executed once on ArangoDB for every database we are using.

We connect to `arangodb` with `arangosh` to issue the following commands two:

```
var graphs = require("@arangodb/general-graph");
graphs._registerCompatibilityFunctions();
```

These have registered all old `GRAPH_*` functions as user-defined functions again, with the prefix `arangodb::` .

Modify the application code

Next we have to go through our application code and replace all calls to `GRAPH_*` by `arangodb::GRAPH_*` . Now run a testrun of our application and check if it worked. If it worked we are ready to go.

Important Information

The user defined functions will call translated subqueries (as described in Solution 2). The optimizer does not know anything about these subqueries beforehand and cannot optimize the whole plan. Also there might be read/write constellations that are forbidden in user-defined functions, therefore a "really" translated query may work while the user-defined function work around may be rejected.

Solution 2: Foxx (recommended)

The general graph module still offers the measurement functions. As these are typically computation expensive and create long running queries it is recommended to not use them in combination with other AQL features. Therefore the best idea is to offer these measurements directly via an API using FOXX.

First we create a new Foxx service. Then we include the `general-graph` module in the service. For every measurement we need we simply offer a GET route to read this measurement.

As an example we do the `GRAPH_RADIUS` :

```
/// ADD FOXX CODE ABOVE

const joi = require('joi');
```

```
const createRouter = require('@arangodb/foxx/router');
const dd = require('dedent');
const router = createRouter();

const graphs = require("@arangodb/general-graph");

router.get('/radius/:graph', function(req, res) {
  let graph;

  // Load the graph
  try {
    graph = graphs._graph(req.graph);
  } catch (e) {
    res.throw('not found');
  }
  res.json(graphs._radius()); // Return the radius
})
.pathParam('graph', joi.string().required(), 'The name of the graph')
.error('not found', 'Graph with this name does not exist.')
.summary('Compute the Radius')
.description(dd`
  This function computes the radius of the given graph
  and returns it.
`);
```

Author: Michael Hackstein

Tags: #howto #aql #migration

Graph

- Fulldepth Graph-Traversal
- Using a custom Visitor
- Example AQL Queries for Graphs

Fulldepth Graph-Traversal

Problem

Lets assume you have a database and some edges and vertices. Now you need the node with the most connections in fulldepth.

Solution

You need a custom traversal with the following properties:

- Store all vertices you have visited already
- If you visit an already visited vertex return the connections + 1 and do not touch the edges
- If you visit a fresh vertex visit all its children and sum up their connections. Store this sum and return it + 1
- Repeat for all vertices.

```
var traversal = require("org/arangodb/graph/traversal");

var knownFilter = function(config, vertex, path) {
  if (config.known[vertex._key] !== undefined) {
    return "prune";
  }
  return "";
};

var sumVisitor = function(config, result, vertex, path) {
  if (config.known[vertex._key] !== undefined) {
    result.sum += config.known[vertex._key];
  } else {
    config.known[vertex._key] = result.sum;
  }
  result.sum += 1;
  return;
};

var config = {
  datasource: traversal.collectionDatasourceFactory(db.e), // e is my edge collection
  strategy: "depthfirst",
  order: "preorder",
  filter: knownFilter,
  expander: traversal.outboundExpander,
  visitor: sumVisitor,
  known: {}
};

var traverser = new traversal.Traverser(config);
var cursor = db.v.all(); // v is my vertex collection
while(cursor.hasNext()) {
  var node = cursor.next();
  traverser.traverse({sum: 0}, node);
}

config.known; // Returns the result of type name: counter. In arangosh this will print out complete result
```

To execute this script accordingly replace db.v and db.e with your collections (v is vertices, e is edges) and write it to a file: (e.g.) traverse.js then execute it in arangosh:

```
cat traverse.js | arangosh
```

If you want to use it in production you should have a look at the Foxx framework which allows you to store and execute this script on server side and make it accessible via your own API: Foxx

Comment

You only compute the connections of one vertex once and cache it then. Complexity is almost equal to the amount of edges. In the code below config.known contains the result of all vertices, you then can add the sorting on it.

Author: Michael Hackstein

Tags: #graph

Using a custom visitor from node.js

Problem

I want to traverse a graph using a custom visitor from node.js.

Solution

Use arangojs and an AQL query with a custom visitor.

Installing arangojs

First thing is to install *arangojs*. This can be done using *npm* or *bower*:

```
npm install arangojs
```

or

```
bower install arangojs
```

Example data setup

For the following example, we need the example graph and data from here. Please download the code from the link and store it in the filesystem using a filename of `world-graph-setup.js`. Then start the ArangoShell and run the code from the file:

```
require("internal").load("/path/to/file/world-graph-setup.js");
```

The script will create the following two collections and load some data into them:

- `v` : a collection with vertex documents
- `e` : an edge collection containing the connections between vertices in `v`

Registering a custom visitor function

Let's register a custom visitor function now. A custom visitor function is a JavaScript function that is executed every time the traversal processes a vertex in the graph.

To register a custom visitor function, we can execute the following commands in the ArangoShell:

```
var aqlfunctions = require("org/arangodb/aql/functions");

aqlfunctions.register("myfunctions::leafNodeVisitor", function (config, result, vertex, path, connected) {
  if (connected && connected.length === 0) {
    return vertex.name + " (" + vertex.type + ")";
  }
});
```

Invoking the custom visitor

The following code can be run in node.js to execute an AQL query that will make use of the custom visitor:

```
Database = require('arangojs');

/* connection the database, change as required */
db = new Database('http://127.0.0.1:8529');

/* the query string */
```

```
var query = "FOR result IN TRAVERSAL(v, e, @vertex, 'inbound', @options) RETURN result";

/* bind parameters */
var bindVars = {
  vertex: "v/world",  /* our start vertex */
  options: {
    order: "preorder-expander",
    visitor: "myfunctions::leafNodeVisitor",
    visitorReturnsResults: true
  }
};

db.query(query, bindVars, function (err, cursor) {
  if (err) {
    console.log('error: %j', err);
  } else {
    cursor.all(function(err2, list) {
      if (err) {
        console.log('error: %j', err2);
      } else {
        console.log("all document keys: %j", list);
      }
    });
  }
});
```

Author: Jan Steemann

Tags: #graph #traversal #aql #nodejs

AQL Example Queries on an Actors and Movies Database

Acknowledgments

On Stackoverflow the user Vincz asked for some example queries based on graphs. So credits for this questions go to him. The datasets and queries have been taken from the guys of neo4j. Credits and thanks to them. As I also think this examples are yet missing I decided to write this recipe.

Problem

(Copy from Stackoverflow) Given a collection of **actors** and a collection of **movies**. And a **actIn** edges collection (with a **year** property) to connect the vertex.

[Actor] ← act in → [Movie]

How could I get:

- All actors who acted in "movie1" OR "movie2"
- All actors who acted in both "movie1" AND "movie2" ?
- All common movies between "actor1" and "actor2" ?
- All actors who acted in 3 or more movies ?
- All movies where exactly 6 actors acted in ?
- The number of actors by movie ?
- The number of movies by actor ?
- The number of movies acted in between 2005 and 2010 by actor ?

Solution

During this solution we will be using arangosh to create and query the data. All the AQL queries are strings and can simply be copied over to your favorite driver instead of arangosh.

Create a Test Dataset in arangosh:

```
var actors = db._create("actors");
var movies = db._create("movies");
var actsIn = db._createEdgeCollection("actsIn");

var TheMatrix = movies.save({_key: "TheMatrix", title:'The Matrix', released:1999, tagline:'Welcome to the Real World'})._id;
var Keanu = actors.save({_key: "Keanu", name:'Keanu Reeves', born:1964})._id;
var Carrie = actors.save({_key: "Carrie", name:'Carrie-Anne Moss', born:1967})._id;
var Laurence = actors.save({_key: "Laurence", name:'Laurence Fishburne', born:1961})._id;
var Hugo = actors.save({_key: "Hugo", name:'Hugo Weaving', born:1960})._id;
var Emil = actors.save({_key: "Emil", name:"Emil Eifrem", born: 1978});

actsIn.save(Keanu, TheMatrix, {roles: ["Neo"], year: 1999});
actsIn.save(Carrie, TheMatrix, {roles: ["Trinity"], year: 1999});
actsIn.save(Laurence, TheMatrix, {roles: ["Morpheus"], year: 1999});
actsIn.save(Hugo, TheMatrix, {roles: ["Agent Smith"], year: 1999});
actsIn.save(Emil, TheMatrix, {roles: ["Emil"], year: 1999});

var TheMatrixReloaded = movies.save({_key: "TheMatrixReloaded", title: "The Matrix Reloaded", released: 2003, tagline: "Free yo
ur mind"});
actsIn.save(Keanu, TheMatrixReloaded, {roles: ["Neo"], year: 2003});
actsIn.save(Carrie, TheMatrixReloaded, {roles: ["Trinity"], year: 2003});
actsIn.save(Laurence, TheMatrixReloaded, {roles: ["Morpheus"], year: 2003});
actsIn.save(Hugo, TheMatrixReloaded, {roles: ["Agent Smith"], year: 2003});

var TheMatrixRevolutions = movies.save({_key: "TheMatrixRevolutions", title: "The Matrix Revolutions", released: 2003, tagline:
"Everything that has a beginning has an end"});
actsIn.save(Keanu, TheMatrixRevolutions, {roles: ["Neo"], year: 2003});
actsIn.save(Carrie, TheMatrixRevolutions, {roles: ["Trinity"], year: 2003});
actsIn.save(Laurence, TheMatrixRevolutions, {roles: ["Morpheus"], year: 2003});
```

```
actsIn.save(Hugo, TheMatrixRevolutions, {roles: ["Agent Smith"], year: 2003});

var TheDevilsAdvocate = movies.save({_key: "TheDevilsAdvocate", title:"The Devil's Advocate", released:1997, tagline:'Evil has its winning ways'})._id;
var Charlize = actors.save({_key: "Charlize", name:'Charlize Theron', born:1975})._id;
var Al = actors.save({_key: "Al", name:'Al Pacino', born:1940})._id;
actsIn.save(Keanu, TheDevilsAdvocate, {roles: ["Kevin Lomax"], year: 1997});
actsIn.save(Charlize, TheDevilsAdvocate, {roles: ["Mary Ann Lomax"], year: 1997});
actsIn.save(Al, TheDevilsAdvocate, {roles: ["John Milton"], year: 1997});

var AFewGoodMen = movies.save({_key: "AFewGoodMen", title:"A Few Good Men", released:1992, tagline:"In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."})._id;
var TomC = actors.save({_key: "TomC", name:'Tom Cruise', born:1962})._id;
var JackN = actors.save({_key: "JackN", name:'Jack Nicholson', born:1937})._id;
var DemiM = actors.save({_key: "DemiM", name:'Demi Moore', born:1962})._id;
var KevinB = actors.save({_key:"KevinB", name:'Kevin Bacon', born:1958})._id;
var KieferS = actors.save({_key:"KieferS", name:'Kiefer Sutherland', born:1966})._id;
var NoahW = actors.save({_key:"NoahW", name:'Noah Wyle', born:1971})._id;
var CubaG = actors.save({_key:"CubaG", name:'Cuba Gooding Jr.', born:1968})._id;
var KevinP = actors.save({_key:"KevinP", name:'Kevin Pollak', born:1957})._id;
var JTW = actors.save({_key:"JTW", name:'J.T. Walsh', born:1943})._id;
var JamesM = actors.save({_key:"JamesM", name:'James Marshall', born:1967})._id;
var ChristopherG = actors.save({_key:"ChristopherG", name:'Christopher Guest', born:1948})._id;
actsIn.save(TomC,AFewGoodMen,{roles:['Lt. Daniel Kaffee'], year: 1992});
actsIn.save(JackN,AFewGoodMen,{roles:['Col. Nathan R. Jessup'], year: 1992});
actsIn.save(DemiM,AFewGoodMen,{roles:['Lt. Cdr. JoAnne Galloway'], year: 1992});
actsIn.save(KevinB,AFewGoodMen,{roles:['Capt. Jack Ross'], year: 1992});
actsIn.save(KieferS,AFewGoodMen,{ roles:['Lt. Jonathan Kendrick'], year: 1992});
actsIn.save(NoahW,AFewGoodMen,{roles:['Cpl. Jeffrey Barnes'], year: 1992});
actsIn.save(CubaG,AFewGoodMen,{ roles:['Cpl. Carl Hammaker'], year: 1992});
actsIn.save(KevinP,AFewGoodMen,{roles:['Lt. Sam Weinberg'], year: 1992});
actsIn.save(JTW,AFewGoodMen,{roles:['Lt. Col. Matthew Andrew Markinson'], year: 1992});
actsIn.save(JamesM,AFewGoodMen,{roles:['Pfc. Louden Downey'], year: 1992});
actsIn.save(ChristopherG,AFewGoodMen,{ roles:['Dr. Stone'], year: 1992});

var TopGun = movies.save({_key:"TopGun", title:"Top Gun", released:1986, tagline:'I feel the need, the need for speed.'})._id;
var KellyM = actors.save({_key:"KellyM", name:'Kelly McGillis', born:1957})._id;
var ValK = actors.save({_key:"ValK", name:'Val Kilmer', born:1959})._id;
var AnthonyE = actors.save({_key:"AnthonyE", name:'Anthony Edwards', born:1962})._id;
var TomS = actors.save({_key:"TomS", name:'Tom Skerritt', born:1933})._id;
var MegR = actors.save({_key:"MegR", name:'Meg Ryan', born:1961})._id;
actsIn.save(TomC,TopGun,{roles:['Maverick'], year: 1986});
actsIn.save(KellyM,TopGun,{roles:['Charlie'], year: 1986});
actsIn.save(ValK,TopGun,{roles:['Iceman'], year: 1986});
actsIn.save(AnthonyE,TopGun,{roles:['Goose'], year: 1986});
actsIn.save(TomS,TopGun,{roles:['Viper'], year: 1986});
actsIn.save(MegR,TopGun,{roles:['Carole'], year: 1986});

var JerryMaguire = movies.save({_key:"JerryMaguire", title:'Jerry Maguire', released:2000, tagline:'The rest of his life begins now.'})._id;
var ReneeZ = actors.save({_key:"ReneeZ", name:'Renee Zellweger', born:1969})._id;
var KellyP = actors.save({_key:"KellyP", name:'Kelly Preston', born:1962})._id;
var JerryO = actors.save({_key:"JerryO", name:"Jerry O'Connell", born:1974})._id;
var JayM = actors.save({_key:"JayM", name:'Jay Mohr', born:1970})._id;
var BonnieH = actors.save({_key:"BonnieH", name:'Bonnie Hunt', born:1961})._id;
var ReginaK = actors.save({_key:"ReginaK", name:'Regina King', born:1971})._id;
var JonathanL = actors.save({_key:"JonathanL", name:'Jonathan Lipnicki', born:1996})._id;
actsIn.save(TomC,JerryMaguire,{roles:['Jerry Maguire'], year: 2000});
actsIn.save(CubaG,JerryMaguire,{roles:['Rod Tidwell'], year: 2000});
actsIn.save(ReneeZ,JerryMaguire,{roles:['Dorothy Boyd'], year: 2000});
actsIn.save(KellyP,JerryMaguire,{roles:['Avery Bishop'], year: 2000});
actsIn.save(JerryO,JerryMaguire,{roles:['Frank Cushman'], year: 2000});
actsIn.save(JayM,JerryMaguire,{roles:['Bob Sugar'], year: 2000});
actsIn.save(BonnieH,JerryMaguire,{roles:['Laurel Boyd'], year: 2000});
actsIn.save(ReginaK,JerryMaguire,{roles:['Marcee Tidwell'], year: 2000});
actsIn.save(JonathanL,JerryMaguire,{roles:['Ray Boyd'], year: 2000});

var StandByMe = movies.save({_key:"StandByMe", title:"Stand By Me", released:1986, tagline:"For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."})._id;
var RiverP = actors.save({_key:"RiverP", name:'River Phoenix', born:1970})._id;
var CoreyF = actors.save({_key:"CoreyF", name:'Corey Feldman', born:1971})._id;
var WilW = actors.save({_key:"WilW", name:'Wil Wheaton', born:1972})._id;
var JohnC = actors.save({_key:"JohnC", name:'John Cusack', born:1966})._id;
var MarshallB = actors.save({_key:"MarshallB", name:'Marshall Bell', born:1942})._id;
actsIn.save(WilW,StandByMe,{roles:['Gordie Lachance'], year: 1986});
```

```
actsIn.save(RiverP,StandByMe,{roles:['Chris Chambers'], year: 1986});
actsIn.save(JerryO,StandByMe,{roles:['Vern Tessio'], year: 1986});
actsIn.save(CoreyF,StandByMe,{roles:['Teddy Duchamp'], year: 1986});
actsIn.save(JohnC,StandByMe,{roles:['Denny Lachance'], year: 1986});
actsIn.save(KieferS,StandByMe,{roles:['Ace Merrill'], year: 1986});
actsIn.save(MarshallB,StandByMe,{roles:['Mr. Lachance'], year: 1986});

var AsGoodAsItGets = movies.save({_key:"AsGoodAsItGets", title:'As Good as It Gets', released:1997, tagline:'A comedy from the heart that goes for the throat.'})._id;
var HelenH = actors.save({_key:"HelenH", name:'Helen Hunt', born:1963})._id;
var GregK = actors.save({_key:"GregK", name:'Greg Kinnear', born:1963})._id;
actsIn.save(JackN,AsGoodAsItGets,{roles:['Melvin Udall'], year: 1997});
actsIn.save(HelenH,AsGoodAsItGets,{roles:['Carol Connelly'], year: 1997});
actsIn.save(GregK,AsGoodAsItGets,{roles:['Simon Bishop'], year: 1997});
actsIn.save(CubaG,AsGoodAsItGets,{roles:['Frank Sachs'], year: 1997});

var WhatDreamsMayCome = movies.save({_key:"WhatDreamsMayCome", title:'What Dreams May Come', released:1998, tagline:'After life there is more. The end is just the beginning.'})._id;
var AnnabellaS = actors.save({_key:"AnnabellaS", name:'Annabella Sciorra', born:1960})._id;
var MaxS = actors.save({_key:"MaxS", name:'Max von Sydow', born:1929})._id;
var WernerH = actors.save({_key:"WernerH", name:'Werner Herzog', born:1942})._id;
var Robin = actors.save({_key:"Robin", name:'Robin Williams', born:1951})._id;
actsIn.save(Robin,WhatDreamsMayCome,{roles:['Chris Nielsen'], year: 1998});
actsIn.save(CubaG,WhatDreamsMayCome,{roles:['Albert Lewis'], year: 1998});
actsIn.save(AnnabellaS,WhatDreamsMayCome,{roles:['Annie Collins-Nielsen'], year: 1998});
actsIn.save(MaxS,WhatDreamsMayCome,{roles:['The Tracker'], year: 1998});
actsIn.save(WernerH,WhatDreamsMayCome,{roles:['The Face'], year: 1998});

var SnowFallingonCedars = movies.save({_key:"SnowFallingonCedars", title:'Snow Falling on Cedars', released:1999, tagline:'First loves last. Forever.'})._id;
var EthanH = actors.save({_key:"EthanH", name:'Ethan Hawke', born:1970})._id;
var RickY = actors.save({_key:"RickY", name:'Rick Yune', born:1971})._id;
var JamesC = actors.save({_key:"JamesC", name:'James Cromwell', born:1940})._id;
actsIn.save(EthanH,SnowFallingonCedars,{roles:['Ishmael Chambers'], year: 1999});
actsIn.save(RickY,SnowFallingonCedars,{roles:['Kazuo Miyamoto'], year: 1999});
actsIn.save(MaxS,SnowFallingonCedars,{roles:['Nels Gudmundsson'], year: 1999});
actsIn.save(JamesC,SnowFallingonCedars,{roles:['Judge Fielding'], year: 1999});

var YouveGotMail = movies.save({_key:"YouveGotMail", title:"You've Got Mail", released:1998, tagline:'At odds in life... in love on-line.'})._id;
var ParkerP = actors.save({_key:"ParkerP", name:'Parker Posey', born:1968})._id;
var DaveC = actors.save({_key:"DaveC", name:'Dave Chappelle', born:1973})._id;
var SteveZ = actors.save({_key:"SteveZ", name:'Steve Zahn', born:1967})._id;
var TomH = actors.save({_key:"TomH", name:'Tom Hanks', born:1956})._id;
actsIn.save(TomH,YouveGotMail,{roles:['Joe Fox'], year: 1998});
actsIn.save(MegR,YouveGotMail,{roles:['Kathleen Kelly'], year: 1998});
actsIn.save(GregK,YouveGotMail,{roles:['Frank Navasky'], year: 1998});
actsIn.save(ParkerP,YouveGotMail,{roles:['Patricia Eden'], year: 1998});
actsIn.save(DaveC,YouveGotMail,{roles:['Kevin Jackson'], year: 1998});
actsIn.save(SteveZ,YouveGotMail,{roles:['George Pappas'], year: 1998});

var SleeplessInSeattle = movies.save({_key:"SleeplessInSeattle", title:'Sleepless in Seattle', released:1993, tagline:'What if someone you never met, someone you never saw, someone you never knew was the only someone for you?'})._id;
var RitaW = actors.save({_key:"RitaW", name:'Rita Wilson', born:1956})._id;
var BillPull = actors.save({_key:"BillPull", name:'Bill Pullman', born:1953})._id;
var VictorG = actors.save({_key:"VictorG", name:'Victor Garber', born:1949})._id;
var RosieO = actors.save({_key:"RosieO", name:"Rosie O'Donnell", born:1962})._id;
actsIn.save(TomH,SleeplessInSeattle,{roles:['Sam Baldwin'], year: 1993});
actsIn.save(MegR,SleeplessInSeattle,{roles:['Annie Reed'], year: 1993});
actsIn.save(RitaW,SleeplessInSeattle,{roles:['Suzy'], year: 1993});
actsIn.save(BillPull,SleeplessInSeattle,{roles:['Walter'], year: 1993});
actsIn.save(VictorG,SleeplessInSeattle,{roles:['Greg'], year: 1993});
actsIn.save(RosieO,SleeplessInSeattle,{roles:['Becky'], year: 1993});

var JoeVersustheVolcano = movies.save({_key:"JoeVersustheVolcano", title:'Joe Versus the Volcano', released:1990, tagline:'A story of love, lava and burning desire.'})._id;
var Nathan = actors.save({_key:"Nathan", name:'Nathan Lane', born:1956})._id;
actsIn.save(TomH,JoeVersustheVolcano,{roles:['Joe Banks'], year: 1990});
actsIn.save(MegR,JoeVersustheVolcano,{roles:['DeDe', 'Angelica Graynamore', 'Patricia Graynamore'], year: 1990});
actsIn.save(Nathan,JoeVersustheVolcano,{roles:['Baw'], year: 1990});

var WhenHarryMetSally = movies.save({_key:"WhenHarryMetSally", title:'When Harry Met Sally', released:1998, tagline:'At odds in life... in love on-line.'})._id;
var BillyC = actors.save({_key:"BillyC", name:'Billy Crystal', born:1948})._id;
var CarrieF = actors.save({_key:"CarrieF", name:'Carrie Fisher', born:1956})._id;
var BrunoK = actors.save({_key:"BrunoK", name:'Bruno Kirby', born:1949})._id;
```

```
actsIn.save(BillyC,WhenHarryMetSally,{roles:['Harry Burns'], year: 1998});
actsIn.save(MegR,WhenHarryMetSally,{roles:['Sally Albright'], year: 1998});
actsIn.save(CarrieF,WhenHarryMetSally,{roles:['Marie'], year: 1998});
actsIn.save(BrunoK,WhenHarryMetSally,{roles:['Jess'], year: 1998});
```

All actors who acted in "movie1" OR "movie2"

Say we want to find all actors who acted in "TheMatrix" OR "TheDevilsAdvocate":

First lets try to get all actors for one movie:

```
db._query("FOR x IN ANY 'movies/TheMatrix' actsIn OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN x._id").toArray();
```

Result:

```
[
  [
    "actors/Keanu",
    "actors/Hugo",
    "actors/Emil",
    "actors/Carrie",
    "actors/Laurence"
  ]
]
```

Now we continue to form a UNION_DISTINCT of two NEIGHBORS queries which will be the solution:

```
db._query("FOR x IN UNION_DISTINCT ((FOR y IN ANY 'movies/TheMatrix' actsIn OPTIONS {bfs: true, uniqueVertices: 'global'} RETUR
N y._id), (FOR y IN ANY 'movies/TheDevilsAdvocate' actsIn OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN y._id)) RETURN x"
).toArray();
```

```
[
  [
    "actors/Emil",
    "actors/Hugo",
    "actors/Carrie",
    "actors/Laurence",
    "actors/Keanu",
    "actors/Al",
    "actors/Charlize"
  ]
]
```

All actors who acted in both "movie1" AND "movie2" ?

This is almost identical to the question above. But this time we are not intrested in a UNION but in a INTERSECTION:

```
db._query("FOR x IN INTERSECTION ((FOR y IN ANY 'movies/TheMatrix' actsIn OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN
y._id), (FOR y IN ANY 'movies/TheDevilsAdvocate' actsIn OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN y._id)) RETURN x")
.toArray();
```

```
[
  [
    "actors/Keanu"
  ]
]
```

All common movies between "actor1" and "actor2" ?

This is actually identical to the question about common actors in movie1 and movie2. We just have to change the starting vertices. As an example let's find all movies where Hugo Weaving ("Hugo") and Keanu Reeves are co-starring:

```
db._query("FOR x IN INTERSECTION ((FOR y IN ANY 'actors/Hugo' actsIn OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN y._id
), (FOR y IN ANY 'actors/Keanu' actsIn OPTIONS {bfs: true, uniqueVertices: 'global'} RETURN y._id)) RETURN x").toArray();
```

```
[
  [
    "movies/TheMatrixRevolutions",
    "movies/TheMatrixReloaded",
    "movies/TheMatrix"
  ]
]
```

All actors who acted in 3 or more movies ?

This question is different, we cannot make use of the neighbors function here. Instead we will make use of the edge-index and the COLLECT statement of AQL for grouping. The basic idea is to group all edges by their startVertex (which in this dataset is always the actor). Then we remove all actors with less than 3 movies from the result. As I am also interested in the number of movies an actor has acted in, I included the value in the result as well:

```
db._query("FOR x IN actsIn COLLECT actor = x._from WITH COUNT INTO counter FILTER counter >= 3 RETURN {actor: actor, movies: co
unter}").toArray()
```

```
[
  {
    "actor" : "actors/Carrie",
    "movies" : 3
  },
  {
    "actor" : "actors/CubaG",
    "movies" : 4
  },
  {
    "actor" : "actors/Hugo",
    "movies" : 3
  },
  {
    "actor" : "actors/Keanu",
    "movies" : 4
  },
  {
    "actor" : "actors/Laurence",
    "movies" : 3
  },
  {
    "actor" : "actors/MegR",
    "movies" : 5
  },
  {
    "actor" : "actors/TomC",
    "movies" : 3
  },
  {
    "actor" : "actors/TomH",
    "movies" : 3
  }
]
```

All movies where exactly 6 actors acted in ?

The same idea as in the query before, but with equality filter, however now we need the movie instead of the actor, so we return the _to attribute:

```
db._query("FOR x IN actsIn COLLECT movie = x._to WITH COUNT INTO counter FILTER counter == 6 RETURN movie").toArray()
```


```
[
  "movies/SleeplessInSeattle",
  "movies/TopGun",
  "movies/YouveGotMail"
]
```


The number of actors by movie ?

We remember in our dataset _to on the edge corresponds to the movie, so we count how often the same _to appears. This is the number of actors. The query is almost identical to the ones before but without the FILTER after COLLECT:

```
db._query("FOR x IN actsIn COLLECT movie = x._to WITH COUNT INTO counter RETURN {movie: movie, actors: counter}").toArray()
```


```
[
  {
    "movie" : "movies/AFewGoodMen",
    "actors" : 11
  },
  {
    "movie" : "movies/AsGoodAsItGets",
    "actors" : 4
  },
  {
    "movie" : "movies/JerryMaguire",
    "actors" : 9
  },
  {
    "movie" : "movies/JoeVersustheVolcano",
    "actors" : 3
  },
  {
    "movie" : "movies/SleeplessInSeattle",
    "actors" : 6
  },
  {
    "movie" : "movies/SnowFallingonCedars",
    "actors" : 4
  },
  {
    "movie" : "movies/StandByMe",
    "actors" : 7
  },
  {
    "movie" : "movies/TheDevilsAdvocate",
    "actors" : 3
  },
  {
    "movie" : "movies/TheMatrix",
    "actors" : 5
  },
  {
    "movie" : "movies/TheMatrixReloaded",
    "actors" : 4
  },
  {
    "movie" : "movies/TheMatrixRevolutions",
    "actors" : 4
  },
  {
    "movie" : "movies/TopGun",
    "actors" : 6
  },
  {
    "movie" : "movies/WhatDreamsMayCome",
    "actors" : 5
  },
  {
```

```
    "movie" : "movies/WhenHarryMetSally",
    "actors" : 4
  },
  {
    "movie" : "movies/YouveGotMail",
    "actors" : 6
  }
]
```

The number of movies by actor ?

I think you get the picture by now ;)

```
db._query("FOR x IN actsIn COLLECT actor = x._from WITH COUNT INTO counter RETURN {actor: actor, movies: counter}").toArray()
```

```
[
  {
    "actor" : "actors/Al",
    "movies" : 1
  },
  {
    "actor" : "actors/AnnabellaS",
    "movies" : 1
  },
  {
    "actor" : "actors/AnthonyE",
    "movies" : 1
  },
  {
    "actor" : "actors/BillPull",
    "movies" : 1
  },
  {
    "actor" : "actors/BillyC",
    "movies" : 1
  },
  {
    "actor" : "actors/BonnieH",
    "movies" : 1
  },
  {
    "actor" : "actors/BrunoK",
    "movies" : 1
  },
  {
    "actor" : "actors/Carrie",
    "movies" : 3
  },
  {
    "actor" : "actors/CarrieF",
    "movies" : 1
  },
  {
    "actor" : "actors/Charlize",
    "movies" : 1
  },
  {
    "actor" : "actors/ChristopherG",
    "movies" : 1
  },
  {
    "actor" : "actors/CoreyF",
    "movies" : 1
  },
  {
    "actor" : "actors/CubaG",
    "movies" : 4
  },
  {
    "actor" : "actors/DaveC",
    "movies" : 1
```

```
  },
  {
    "actor" : "actors/DemiM",
    "movies" : 1
  },
  {
    "actor" : "actors/Emil",
    "movies" : 1
  },
  {
    "actor" : "actors/EthanH",
    "movies" : 1
  },
  {
    "actor" : "actors/GregK",
    "movies" : 2
  },
  {
    "actor" : "actors/HelenH",
    "movies" : 1
  },
  {
    "actor" : "actors/Hugo",
    "movies" : 3
  },
  {
    "actor" : "actors/JackN",
    "movies" : 2
  },
  {
    "actor" : "actors/JamesC",
    "movies" : 1
  },
  {
    "actor" : "actors/JamesM",
    "movies" : 1
  },
  {
    "actor" : "actors/JayM",
    "movies" : 1
  },
  {
    "actor" : "actors/JerryO",
    "movies" : 2
  },
  {
    "actor" : "actors/JohnC",
    "movies" : 1
  },
  {
    "actor" : "actors/JonathanL",
    "movies" : 1
  },
  {
    "actor" : "actors/JTW",
    "movies" : 1
  },
  {
    "actor" : "actors/Keanu",
    "movies" : 4
  },
  {
    "actor" : "actors/KellyM",
    "movies" : 1
  },
  {
    "actor" : "actors/KellyP",
    "movies" : 1
  },
  {
    "actor" : "actors/KevinB",
    "movies" : 1
  },
  {
    "actor" : "actors/KevinP",
    "movies" : 1
```

```
  },
  {
    "actor" : "actors/KieferS",
    "movies" : 2
  },
  {
    "actor" : "actors/Laurence",
    "movies" : 3
  },
  {
    "actor" : "actors/MarshallB",
    "movies" : 1
  },
  {
    "actor" : "actors/MaxS",
    "movies" : 2
  },
  {
    "actor" : "actors/MegR",
    "movies" : 5
  },
  {
    "actor" : "actors/Nathan",
    "movies" : 1
  },
  {
    "actor" : "actors/NoahW",
    "movies" : 1
  },
  {
    "actor" : "actors/ParkerP",
    "movies" : 1
  },
  {
    "actor" : "actors/ReginaK",
    "movies" : 1
  },
  {
    "actor" : "actors/ReneeZ",
    "movies" : 1
  },
  {
    "actor" : "actors/RickY",
    "movies" : 1
  },
  {
    "actor" : "actors/RitaW",
    "movies" : 1
  },
  {
    "actor" : "actors/RiverP",
    "movies" : 1
  },
  {
    "actor" : "actors/Robin",
    "movies" : 1
  },
  {
    "actor" : "actors/RosieO",
    "movies" : 1
  },
  {
    "actor" : "actors/SteveZ",
    "movies" : 1
  },
  {
    "actor" : "actors/TomC",
    "movies" : 3
  },
  {
    "actor" : "actors/TomH",
    "movies" : 3
  },
  {
    "actor" : "actors/TomS",
    "movies" : 1
```

```
  },
  {
    "actor" : "actors/ValK",
    "movies" : 1
  },
  {
    "actor" : "actors/VictorG",
    "movies" : 1
  },
  {
    "actor" : "actors/WernerH",
    "movies" : 1
  },
  {
    "actor" : "actors/WilW",
    "movies" : 1
  }
]
```

The number of movies acted in between 2005 and 2010 by actor ?

This query is where a Multi Model database actually shines. First of all we want to use it in production, so we set a skiplistindex on year. This allows as to execute fast range queries like between 2005 and 2010.

```
db.actsIn.ensureSkiplist("year")
```

Now we slightly modify our movies by actor query. However my dataset contains only older movies, so I changed the year range from 1990 - 1995:

```
db._query("FOR x IN actsIn FILTER x.year >= 1990 && x.year <= 1995 COLLECT actor = x._from WITH COUNT INTO counter RETURN {acto
r: actor, movies: counter}").toArray()
```

```
[
  {
    "actor" : "actors/BillPull",
    "movies" : 1
  },
  {
    "actor" : "actors/ChristopherG",
    "movies" : 1
  },
  {
    "actor" : "actors/CubaG",
    "movies" : 1
  },
  {
    "actor" : "actors/DemiM",
    "movies" : 1
  },
  {
    "actor" : "actors/JackN",
    "movies" : 1
  },
  {
    "actor" : "actors/JamesM",
    "movies" : 1
  },
  {
    "actor" : "actors/JTW",
    "movies" : 1
  },
  {
    "actor" : "actors/KevinB",
    "movies" : 1
  },
  {
    "actor" : "actors/KevinP",
    "movies" : 1
  },
```

```
  {
    "actor" : "actors/KieferS",
    "movies" : 1
  },
  {
    "actor" : "actors/MegR",
    "movies" : 2
  },
  {
    "actor" : "actors/Nathan",
    "movies" : 1
  },
  {
    "actor" : "actors/NoahW",
    "movies" : 1
  },
  {
    "actor" : "actors/RitaW",
    "movies" : 1
  },
  {
    "actor" : "actors/RosieO",
    "movies" : 1
  },
  {
    "actor" : "actors/TomC",
    "movies" : 1
  },
  {
    "actor" : "actors/TomH",
    "movies" : 2
  },
  {
    "actor" : "actors/VictorG",
    "movies" : 1
  }
]
```


Comment

Author: Michael Hackstein

Tags: #graph #examples

Use Cases / Examples

- Crawling Github with Promises
- Using ArangoDB with Sails.js
- Populating a Textbox
- Exporting Data
- Accessing base documents with Java
- Add XML data to ArangoDB with Java

Crawling Github with Promises

Problem

The new ArangoDB Javascript driver no longer imposes any promises implementation. It follows the standard callback pattern with a callback using `err` and `res`.

But what if we want to use a promise library - in this case the most popular one promises? Lets give it a try and build a **github crawler** with the new Javascript driver and promises.

Solution

The following source code can be found on github.

Pagination with Promises made easy

The github driver has a function to get all followers. However, the result is paginated. With two helper functions and promises it is straight forward to implement a function to retrieve all followers of an user.

```
function extractFollowers (name) {
  'use strict';

  return new Promise(function(resolve, reject) {
    github.user.getFollowers({ user: name }, promoteError(reject, function(res) {
      followPages(resolve, reject, [], res);
    }));
  });
}
```

The `followPages` function simply extends the result with the next page until the last page is reached.

```
function followPages (resolve, reject, result, res) {
  'use strict';

  var i;

  for (i = 0;  i < res.length;  ++i) {
    result.push(res[i]);
  }

  if (github.hasNextPage(res)) {
    github.getNextPage(res, promoteError(reject, function(res) {
      followPages(resolve, reject, result, res);
    }));
  }
  else {
    resolve(result);
  }
}
```

The promote error helper is a convenience function to bridge callbacks and promises.

```
function promoteError (reject, resolve) {
  'use strict';

  return function(err, res) {
    if (err) {
      if (err.hasOwnProperty("message") && /rate limit exceeded/.test(err.message)) {
        rateLimitExceeded = true;
      }

      console.error("caught error: %s", err);
      reject(err);
```

```
    }
    else {
      resolve(res);
    }
  };
}
```

I've decided to stick to the sequence `reject` (aka `err`) followed by `resolve` (aka `res`) - like the callbacks. The `promoteError` can be used for the github callback as well as the ArangoDB driver.

Queues, Queues, Queues

I've only needed a very simple job queue, so queue-it is a good choice. It provides a very simple API for handling job queues:

```
POST /queue/job
POST /queue/worker
DELETE /queue/job/:key
```

The new Javascript driver allows to access arbitrary endpoint. First install a Foxx implementing the queue microservice in an ArangoDB instance.

```
foxx-manager install queue-it /queue
```

Adding a new job from node.js is now easy

```
function addJob (data) {
  'use strict';

  return new Promise(function(resolve, reject) {
    db.endpoint("queue").post("job", data, promoteError(reject, resolve));
  });
}
```

Transaction

I wanted to crawl users and their repos. The relations ("follows", "owns", "is_member", "stars") is stored in an edge collection. I only add an edge if it is not already there. Therefore I check inside a transaction, if the edge exists and add it, if it does not.

```
createRepoDummy(repo.full_name, data).then(function(dummyData) {
  return db.transaction(
    "relations",
    String(function(params) {
      var me = params[0];
      var you = params[1];
      var type = params[2];
      var db = require("org/arangodb").db;

      if (db.relations.firstExample({ _from: me, _to: you, type: type }) === null) {
        db.relations.save(me, you, { type: type });
      }
    }),
    [ meId, "repos/" + data._key, type ],
    function(err) {
      if (err) {
        throw err;
      }

      return handleDummy(dummyData);
    });
})
```

Please note that the action function is executed on the server and not in the nodejs client. Therefore we need to pass the relevant data as parameters. It is not possible to use the closure variables.

Riding the Beast

Start an ArangoDB instance (i.e. inside a docker container) and install the simple queue.

```
foxx-manager install queue-it /queue
```

Start the arangosh and create collections `users`, `repos` and `relations`.

```
arangosh> db._create("users");
arangosh> db.users.ensureHashIndex("name");

arangosh> db._create("repos");
arangosh> db.users.ensureHashIndex("name");

arangosh> db._createEdgeCollection("relations");
```

Now everything is initialized. Fire up nodejs and start crawling:

```
node> var crawler = require("./crawler");
node> crawler.github.authenticate({ type: "basic", username: "username", password: "password" })
node> crawler.addJob({ type:"user", identifier:"username" })
node> crawler.runJobs();
```

Comment

Please keep in mind that this is just an experiment. There is no good error handling and convenience functions for setup and start. It is also not optimized for performance. For instance, it would easily be possible to avoid nodejs / ArangoDB roundtrips using more transactions.

Sources used in this example:

- ArangoJS
- npm promises
- ArangoDB Foxx queue-it

The source code of this example is available from Github: https://github.com/fceller/Foxxmender

Author: Frank Celler

Tags: #foxx #javascript #API #nodejs #driver

How to use ArangoDB with Sails.js

First install the Sails.js framework using NPM:

```
npm install -g sails
```

Now you can create a new Sails.js app named `somename` with the following command:

```
sails new somename
```

Now we need to add ArangoDB to this new application. First `cd` into the freshly created `somename` directory. Then install the ArangoDB adapter for Sails.js with the following command:

```
npm install sails-arangodb
```

This however only installs the necessary dependency. We need to configure the application to load the adapter. Open the file `config/connections.js`. You will see a list of example configurations for possible connections to databases. Remove the ones you don't need (or just keep all of them), and add the following configuration (adjust the host, port, database name and graph name to your needs):

```
localArangoDB: {
  adapter: 'sails-arangodb',
  host: 'localhost',
  port: 8529,
  database: {
    name: 'sails',
    graph: 'sails'
  }
}
```

Now, you can configure your app to use the ArangoDB as your default connection. You do this by adjusting the file `config/models.js`:

```
module.exports.models = {
  connection: 'localArangoDB' // this is the name from the connections.js file
  // ...
};
```

Your app is now configured to use ArangoDB for all models by default. You can now for example create a blueprint controller by typing the following in your console:

```
sails generate api todos
```

Now you can boot your application with:

```
sails lift
```

You can now access `http://localhost:1337/todos` and see an empty list of todos. And then create a todo by visiting `localhost:1337/todos/create?name=john`. This will create the according document (that has an attribute `name` with the value `john`) in the todos collection of the selected database. You will also see the document when you visit `http://localhost:1337/todos` again.

Author: Lucas Dohmen

Tags: #nodejs

Populating an autocomplete textbox

Problem

I want to populate an autocomplete textbox with values from a collection. The completions should adjust dynamically based on user input.

Solution

Use a web framework for the client-side autocomplete rendering and event processing. Use a collection with a (sorted) skiplist index and a range query on it to efficiently fetch the completion values dynamically. Connect the two using a simple Foxx route.

Install an example app

This app contains a jquery-powered web page with an autocomplete textbox. It uses jquery autocomplete, but every other web framework will also do.

The app can be installed as follows:

- in the ArangoDB web interface, switch into the **Applications** tab
- there, click *Add Application*
- switch on the *Github* tab
- for *Repository*, enter `jsteemann/autocomplete`
- for *Version*, enter `master`
- click *Install*

Now enter a mountpoint for the application. This is the URL path under which the application will become available. For the example app, the mountpoint does not matter. The web page in the example app assumes it is served by ArangoDB, too. So it uses a relative URL `autocomplete`. This is easiest to set up, but in reality you might want to have your web page served by a different server. In this case, your web page will have to call the app mountpoint you just entered.

To see the example app in action, click on **Open**. The autocomplete textbox should be populated with server data when at least two letters are entered.

Backend code, setup script

The app also contains a backend route `/autocomplete` which is called by the web page to fetch completions based on user input. The HTML code for the web page is here.

Contained in the app is a setup script that will create a collection named `completions` and load some initial data into it. The example app provides autocompletion for US city names, and the setup script populates the collection with about 10K city names.

The setup script also creates a skiplist index on the lookup attribute, so this attribute can be used for efficient filtering and sorting later. The `lookup` attribute contains the city names already lower-cased, and the original (*pretty*) names are stored in attribute `pretty`. This attribute will be returned to users.

Backend code, Foxx route controller

The app contains a controller. The backend action `/autocomplete` that is called by the web page is also contained herein:

```
controller.get("/autocomplete", function (req, res) {
  // search phrase entered by user
  var searchString = req.params("q").trim() || "";
  // lower bound for search range
  var begin = searchString.replace(/[^a-zA-Z]/g, " ").toLowerCase();
  if (begin.length === 0) {
    // search phrase is empty - no need to perfom a search at all
    res.json([]);
```

```
    return;
  }
  // upper bound for search range
  var end = begin.substr(0, begin.length - 1) + String.fromCharCode(begin.charCodeAt(begin.length - 1) + 1);
  // bind parameters for query
  var queryParams = {
    "@collection" : "completions",
    "begin" : begin,
    "end" : end
  };
  // the search query
  var query = "FOR doc IN @@collection FILTER doc.lookup >= @begin && doc.lookup < @end SORT doc.lookup RETURN { label: doc.pre
tty, value: doc.pretty, id: doc._key }";
  res.json(db._query(query, queryParams).toArray());
}
```

The backend code first fetches the search string from the URL parameter `q`. This is what the web page will send us.

Based on the search string, a lookup range is calculated. First of all, the search string is lower-cased and all non-letter characters are removed from it. The resulting string is the lower bound for the lookup. For the upper bound, we can use the lower bound with its last letter character code increased by one.

For example, if the user entered `Los A` into the textbox, the web page will send us the string `Los A` in URL parameter `q`. Lower-casing and removing non-letter characters from the string, we'll get `losa`. This is the lower bound. The upper bound is `losa`, with its last letter adjusted to `b` (i.e. `losb`).

Finally, the lower and upper bounds are inserted into the following query using bind parameters `@begin` and `@end`:

```
FOR doc IN @@collection
  FILTER doc.lookup >= @begin && doc.lookup < @end
  SORT doc.lookup
  RETURN {
    label: doc.pretty,
    value: doc.pretty,
    id: doc._key
  }
```

The city names in the lookup range will be returned sorted. For each city, three values are returned (the `id` contains the document key, the other two values are for display purposes). Other frameworks may require a different return format, but that can easily be done by adjusting the AQL query.

Author: Jan Steemann

Tags: #aql #autocomplete #jquery

Exporting Data for Offline Processing

In this recipe we will learn how to use the export API to extract data and process it with PHP. At the end of the recipe you can download the complete PHP script.

Note: The following recipe is written using an ArangoDB server with version 2.6 or higher. You can also use the `devel` branch, since version 2.6 hasn't been an official release yet.

Howto

Importing example data

First of all we need some data in an ArangoDB collection. For this example we will use a collection named `users` which we will populate with 100.000 example documents. This way you can get the data into ArangoDB:

```
# download data file
wget https://jsteemann.github.io/downloads/code/users-100000.json.tar.gz
# uncompress it
tar xvfz users-100000.json.tar.gz
# import into ArangoDB
arangoimp --file users-100000.json --collection users --create-collection true
```

Setting up ArangoDB-PHP

For this recipe we will use the ArangoDB PHP driver:

```
git clone -b devel "https://github.com/arangodb/arangodb-php.git"
```

We will now write a simple PHP script that establishes a connection to ArangoDB on localhost:

```
<?php

namespace triagens\ArangoDb;

// use the driver's autoloader to load classes
require 'arangodb-php/autoload.php';
Autoloader::init();

// set up connection options
$connectionOptions = array(
    // endpoint to connect to
    ConnectionOptions::OPTION_ENDPOINT     => 'tcp://localhost:8529',
    // can use Keep-Alive connection
    ConnectionOptions::OPTION_CONNECTION   => 'Keep-Alive',
    // use basic authorization
    ConnectionOptions::OPTION_AUTH_TYPE    => 'Basic',
    // user for basic authorization
    ConnectionOptions::OPTION_AUTH_USER    => 'root',
    // password for basic authorization
    ConnectionOptions::OPTION_AUTH_PASSWD  => '',
    // timeout in seconds
    ConnectionOptions::OPTION_TIMEOUT      => 30,
    // database name
    ConnectionOptions::OPTION_DATABASE     => '_system'
    );

try {
  // establish connection
  $connection = new Connection($connectionOptions);

  echo 'Connected!' . PHP_EOL;

  // TODO: now do something useful with the connection!
```

```
} catch (ConnectException $e) {
  print $e . PHP_EOL;
} catch (ServerException $e) {
  print $e . PHP_EOL;
} catch (ClientException $e) {
  print $e . PHP_EOL;
}
```

After running the script you should see `Connected!` in the bash if successful.

Extracting the data

Now we can run an export of the data in the collection `users`. Place the following code into the `TODO` part of the first code:

```
function export($collection, Connection $connection) {
  $fp = fopen('output.json', 'w');

  if (! $fp) {
    throw new Exception('could not open output file!');
  }

  // settings to use for the export
  $settings = array(
      'batchSize' => 5000,  // export in chunks of 5K documents
      '_flat' => true       // use simple PHP arrays
      );

  $export = new Export($connection, $collection, $settings);

  // execute the export. this will return an export cursor
  $cursor = $export->execute();

  // statistics
  $count   = 0;
  $batches = 0;
  $bytes   = 0;

  // now we can fetch the documents from the collection in batches
  while ($docs = $cursor->getNextBatch()) {
    $output = '';
    foreach ($docs as $doc) {
      $output .= json_encode($doc) . PHP_EOL;
    }

    // write out chunk
    fwrite($fp, $output);

    // update statistics
    $count += count($docs);
    $bytes += strlen($output);
    ++$batches;
  }

  fclose($fp);

  echo sprintf('written %d documents in %d batches with %d total bytes',
      $count,
      $batches,
      $bytes) . PHP_EOL;
}

// run the export
export('users', $connection);
```

The function extracts all documents from the collection and writes them into an output file `output.json`. In addition it will print some statistics about the number of documents and the total data size:

```
written 100000 documents in 20 batches with 40890013 total bytes
```

Applying some transformations

We now will use PHP to transform data as we extract it:

```
function transformDate($value) {
  return preg_replace('/^(\\d+)-(\\d+)-(\\d+)$/', '\\2/\\3/\\1', $value);
}

function transform(array $document) {
  static $genders = array('male' => 'm', 'female' => 'f');

  $transformed = array(
      'gender'      => $genders[$document['gender']],
      'dob'         => transformDate($document['birthday']),
      'memberSince' => transformDate($document['memberSince']),
      'fullName'    => $document['name']['first'] . ' ' . $document['name']['last'],
      'email'       => $document['contact']['email'][0]
      );

  return $transformed;
}

function export($collection, Connection $connection) {
  $fp = fopen('output-transformed.json', 'w');

  if (! $fp) {
    throw new Exception('could not open output file!');
  }

  // settings to use for the export
  $settings = array(
      'batchSize' => 5000,  // export in chunks of 5K documents
      '_flat' => true       // use simple PHP arrays
      );

  $export = new Export($connection, $collection, $settings);

  // execute the export. this will return an export cursor
  $cursor = $export->execute();

  // now we can fetch the documents from the collection in batches
  while ($docs = $cursor->getNextBatch()) {
    $output = '';
    foreach ($docs as $doc) {
      $output .= json_encode(transform($doc)) . PHP_EOL;
    }

    // write out chunk
    fwrite($fp, $output);
  }

  fclose($fp);
}

// run the export
export('users', $connection);
```

With this script the following changes will be made on the data:

- rewrite the contents of the `gender` attribute. `female` becomes `f` and `male` becomes `m`
- `birthday` now becomes `dob`
- the date formations will be changed from YYYY-MM-DD to MM/DD/YYYY
- concatenate the contents of `name.first` and `name.last`
- `contact.email` will be transformed from an array to a flat string
- every other attribute will be removed

Note: The output will be in a file named `output-transformed.json`.

Filtering attributes

Exclude certain attributes

Instead of filtering out as done in the previous example we can easily configure the export to exclude these attributes server-side:

```
// settings to use for the export
$settings = array(
  'batchSize' => 5000,  // export in chunks of 5K documents
  '_flat' => true,      // use simple PHP arrays
  'restrict' => array(
    'type' => 'exclude',
    'fields' => array('_id', '_rev', '_key', 'likes')
  )
);
```

This script will exclude the attributes `_id`, `_rev`. `_key` and `likes`.

Include certain attributes

We can also include attributes with the following script:

```
function export($collection, Connection $connection) {
  // settings to use for the export
  $settings = array(
      'batchSize' => 5000,  // export in chunks of 5K documents
      '_flat' => true,      // use simple PHP arrays
      'restrict' => array(
        'type' => 'include',
        'fields' => array('_key', 'name')
        )
      );

  $export = new Export($connection, $collection, $settings);

  // execute the export. this will return an export cursor
  $cursor = $export->execute();

  // now we can fetch the documents from the collection in batches
  while ($docs = $cursor->getNextBatch()) {
    $output = '';

    foreach ($docs as $doc) {
      $values = array(
          $doc['_key'],
          $doc['name']['first'] . ' ' . $doc['name']['last']
          );

      $output .= '"' . implode('","', $values) . '"' . PHP_EOL;
    }

    // print out the data directly
    print $output;
  }
}

// run the export
export('users', $connection);
```

In this script only the `_key` and `name` attributes are extracted. In the prints the `_key` / `name` pairs are in CSV format.

Note: The whole script can be downloaded.

Using the API without PHP

The export API REST interface can be used with any client that can speak HTTP like curl. With the following command you can fetch the documents from the `users` collection:

```
curl
  -X POST
  http://localhost:8529/_api/export?collection=users
--data '{"batchSize":5000}'
```

The HTTP response will contatin a `result` attribute that contains the actual documents. The attribute `hasMore` will indicate if there are more documents for the client to fetch. The HTTP will also contain an attribute `id` if set to *true*.

With the `id` you can send follow-up requests like this:

```
curl
  -X PUT
  http://localhost:8529/_api/export/13979338067709
```

Authors: Thomas Schmidts and Jan Steemann

Tags: #howto #php

How to retrieve documents from ArangoDB without knowing the structure?

Problem

If you use a NoSQL database it's common to retrieve documents with an unknown attribute structure. Furthermore, the amount and types of attributes may differ in documents resulting from a single query. Another problem is that you want to add one ore more attributes to a document.

In Java you are used to work with objects. Regarding the upper requirements it is possible to directly retrieve objects with the same attribute structure as the document out of the database. Adding attributes to an object at runtime could be very messy.

Note: ArangoDB 3.1 and the corresponding Java driver is needed.

Solution

With the latest version of the Java driver of ArangoDB an object called `BaseDocument` is provided.

The structure is very simple: It only has four attributes:

```
public class BaseDocument {

    String id;
    String key;
    String revision;
    Map<String, Object> properties;

}
```

The first three attributes are the system attributes `_id`, `_key` and `_rev`. The fourth attribute is a `HashMap`. The key always is a String, the value an object. These properties contain all non system attributes of the document.

The map can contain values of the following types:

- Map
- List
- Boolean
- Number
- String
- null

Note: `Map` and `List` contain objects, which are of the same types as listed above.

To retrieve a document is similar to the known procedure, except that you use `BaseDocument` as type.

```
ArangoDB.Builder arango = new ArangoDB.Builder().builder();
DocumentEntity<BaseDocument> myObject = arango.db().collection("myCollection").getDocument("myDocumentKey", BaseDocument.class)
;
```

Other resources

More documentation about the ArangoDB Java driver is available:

- Tutorial: Java in ten minutes
- Java driver at Github
- Example source code
- JavaDoc

Author: gschwab, Mark Vollmary

Tags: #java #driver

How to add XML data to ArangoDB?

Problem

You want to store XML data files into a database to have the ability to make queries onto them.

Note: ArangoDB 3.1 and the corresponding Java driver is needed.

Solution

Since version 3.1.0 the aragodb-java-driver supports writing, reading and querying of raw strings containing the JSON documents.

With JsonML you can convert a XML string into a JSON string and back to XML again.

Converting XML into JSON with JsonML example:

```
String string = "<recipe name=\"bread\" prep_time=\"5 mins\" cook_time=\"3 hours\"> "
        + "<title>Basic bread</title> "
        + "<ingredient amount=\"8\" unit=\"dL\">Flour</ingredient> "
        + "<ingredient amount=\"10\" unit=\"grams\">Yeast</ingredient> "
        + "<ingredient amount=\"4\" unit=\"dL\" state=\"warm\">Water</ingredient> "
        + "<ingredient amount=\"1\" unit=\"teaspoon\">Salt</ingredient> "
        + "<instructions> "
        + "<step>Mix all ingredients together.</step> "
        + "<step>Knead thoroughly.</step> "
        + "<step>Cover with a cloth, and leave for one hour in warm room.</step> "
        + "<step>Knead again.</step> "
        + "<step>Place in a bread baking tin.</step> "
        + "<step>Cover with a cloth, and leave for one hour in warm room.</step> "
        + "<step>Bake in the oven at 180(degrees)C for 30 minutes.</step> "
        + "</instructions> "
        + "</recipe> ";

JSONObject jsonObject = JSONML.toJSONObject(string);
System.out.println(jsonObject.toString());
```

The converted JSON string:

```
{
   "prep_time" : "5 mins",
   "name" : "bread",
   "cook_time" : "3 hours",
   "tagName" : "recipe",
   "childNodes" : [
      {
         "childNodes" : [
            "Basic bread"
         ],
         "tagName" : "title"
      },
      {
         "childNodes" : [
            "Flour"
         ],
         "tagName" : "ingredient",
         "amount" : 8,
         "unit" : "dL"
      },
      {
         "unit" : "grams",
         "amount" : 10,
         "tagName" : "ingredient",
         "childNodes" : [
            "Yeast"
         ]
      },
```

```
      {
         "childNodes" : [
            "Water"
         ],
         "tagName" : "ingredient",
         "amount" : 4,
         "unit" : "dL",
         "state" : "warm"
      },
      {
         "childNodes" : [
            "Salt"
         ],
         "tagName" : "ingredient",
         "unit" : "teaspoon",
         "amount" : 1
      },
      {
         "childNodes" : [
            {
               "tagName" : "step",
               "childNodes" : [
                  "Mix all ingredients together."
               ]
            },
            {
               "tagName" : "step",
               "childNodes" : [
                  "Knead thoroughly."
               ]
            },
            {
               "childNodes" : [
                  "Cover with a cloth, and leave for one hour in warm room."
               ],
               "tagName" : "step"
            },
            {
               "tagName" : "step",
               "childNodes" : [
                  "Knead again."
               ]
            },
            {
               "childNodes" : [
                  "Place in a bread baking tin."
               ],
               "tagName" : "step"
            },
            {
               "tagName" : "step",
               "childNodes" : [
                  "Cover with a cloth, and leave for one hour in warm room."
               ]
            },
            {
               "tagName" : "step",
               "childNodes" : [
                  "Bake in the oven at 180(degrees)C for 30 minutes."
               ]
            }
         ],
         "tagName" : "instructions"
      }
   ]
}
```

Saving the converted JSON to ArangoDB example:

```
ArangoDB.Builder arango = new ArangoDB.Builder().build();
ArangoCollection collection = arango.db().collection("testCollection")
DocumentCreateEntity<String> entity = collection.insertDocument(
                jsonObject.toString());
String key = entity.getKey();
```

Reading the stored JSON as a string and convert it back to XML example:

```
String rawJsonString = collection.getDocument(key, String.class);
String xml = JSONML.toString(rawJsonString);
System.out.println(xml);
```

Example output:

```
<recipe _id="RawDocument/6834407522" _key="6834407522" _rev="6834407522"
         cook_time="3 hours" name="bread" prep_time="5 mins">
  <title>Basic bread</title>
  <ingredient amount="8" unit="dL">Flour</ingredient>
  <ingredient amount="10" unit="grams">Yeast</ingredient>
  <ingredient amount="4" state="warm" unit="dL">Water</ingredient>
  <ingredient amount="1" unit="teaspoon">Salt</ingredient>
  <instructions>
    <step>Mix all ingredients together.</step>
    <step>Knead thoroughly.</step>
    <step>Cover with a cloth, and leave for one hour in warm room.</step>
    <step>Knead again.</step>
    <step>Place in a bread baking tin.</step>
    <step>Cover with a cloth, and leave for one hour in warm room.</step>
    <step>Bake in the oven at 180(degrees)C for 30 minutes.</step>
  </instructions>
</recipe>
```

Note: The fields mandatory to ArangoDB documents are added; If they break your XML schema you have to remove them.

Query raw data example:

```
String queryString = "FOR t IN testCollection FILTER t.cook_time == '3 hours' RETURN t";
ArangoCursor<String> cursor = arango.db().query(queryString, null, null, String.class);
while (cursor.hasNext()) {
    JSONObject jsonObject = new JSONObject(cursor.next());
    String xml = JSONML.toString(jsonObject);
    System.out.println("XML value: " + xml);
}
```

Other resources

More documentation about the ArangoDB Java driver is available:

- Tutorial: Java in ten minutes
- Java driver at Github
- Example source code
- JavaDoc

Author: Achim Brandt, Mark Vollmary

Tags: #java #driver

Administration

- Using Authentication
- Importing Data
- Replicating Data
- XCopy Install Windows
- Migrating 2.8 to 3.0

Using authentication

Problem

I want to use authentication in ArangoDB.

Solution

In order to make authentication work properly, you will need to create user accounts first.

Then adjust ArangoDB's configuration and turn on authentication (if it's off).

Set up or adjust user accounts

ArangoDB user accounts are valid throughout a server instance and users can be granted access to one or more databases. They are managed through the database named `_system`.

To manage user accounts, connect with the ArangoShell to the ArangoDB host and the `_system` database:

```
$ arangosh --server.endpoint tcp://127.0.0.1:8529 --server.database "_system"
```

By default, arangosh will connect with a username `root` and an empty password. This will work if authentication is turned off.

When connected, you can create a new user account with the following command:

```
arangosh> require("org/arangodb/users").save("myuser", "mypasswd");
```

`myuser` will be the username and `mypasswd` will be the user's password. Note that running the command like this may store the password literally in ArangoShell's history.

To avoid that, use a dynamically created password, e.g.:

```
arangosh> passwd = require("internal").genRandomAlphaNumbers(20);
arangosh> require("org/arangodb/users").save("myuser", passwd);
```

The above will print the password on screen (so you can memorize it) but won't store it in the command history.

While there, you probably want to change the password of the default `root` user too. Otherwise one will be able to connect with the default `root` user and its empty password. The following commands change the `root` user's password:

```
arangosh> passwd = require("internal").genRandomAlphaNumbers(20);
arangosh> require("org/arangodb/users").update("root", passwd);
```

Turn on authentication

Authentication is turned on by default in ArangoDB. You should make sure that it was not turned off manually however. Check the configuration file (normally named `/etc/arangodb.conf`) and make sure it contains the following line in the `server` section:

```
authentication = true
```

This will make ArangoDB require authentication for every request (including requests to Foxx apps).

If you want to run Foxx apps without HTTP authentcation, but activate HTTP authentication for the built-in server APIs, you can add the following line in the `server` section of the configuration:

```
authentication-system-only = true
```

The above will bypass authentication for requests to Foxx apps.

When finished making changes, you need to restart ArangoDB:

```
service arangodb restart
```

Check accessibility

To confirm authentication is in effect, try connecting to ArangoDB with the ArangoShell:

```
$ arangosh --server.endpoint tcp://127.0.0.1:8529 --server.database "_system"
```

The above will implicity use a username `root` and an empty password when connecting. If you changed the password of the `root` account as described above, this should not work anymore.

You should also validate that you can connect with a valid user:

```
$ arangosh --server.endpoint tcp://127.0.0.1:8529 --server.database "_system" --server.username myuser
```

You can also use curl to check that you are actually getting HTTP 401 (Unauthorized) server responses for requests that require authentication:

```
$ curl --dump - http://127.0.0.1:8529/_api/version
```

Author: Jan Steemann

Tags: #authentication #security

Importing data

Problem

I want to import data from a file into ArangoDB.

Solution

ArangoDB comes with a command-line tool utility named `arangoimp`. This utility can be used for importing JSON-encoded, CSV, and tab-separated files into ArangoDB.

`arangoimp` needs to be invoked from the command-line once for each import file. The target collection can already exist or can be created by the import run.

Importing JSON-encoded data

Input formats

There are two supported input formats for importing JSON-encoded data into ArangoDB:

- **line-by-line format**: This format expects each line in the input file to be a valid JSON objects. No line breaks must occur within each single JSON object
- **array format**: Expects a file containing a single array of JSON objects. Whitespace is allowed for formatting inside the JSON array and the JSON objects

Here's an example for the **line-by-line format** looks like this:

```
{"author":"Frank Celler","time":"2011-10-26 08:42:49 +0200","sha":"c413859392a45873936cbe40797970f8eed93ff9","message":"first c
ommit","user":"f.celler"}
{"author":"Frank Celler","time":"2011-10-26 21:32:36 +0200","sha":"10bb77b8cc839201ff59a778f0c740994083c96e","message":"initial
 release","user":"f.celler"}
...
```

Here's an example for the same data in **array format**:

```
[
  {
    "author": "Frank Celler",
    "time": "2011-10-26 08:42:49 +0200",
    "sha": "c413859392a45873936cbe40797970f8eed93ff9",
    "message": "first commit",
    "user": "f.celler"
  },
  {
    "author": "Frank Celler",
    "time": "2011-10-26 21:32:36 +0200",
    "sha": "10bb77b8cc839201ff59a778f0c740994083c96e",
    "message": "initial release",
    "user": "f.celler"
  },
  ...
]
```

Importing JSON data in line-by-line format

An example data file in **line-by-line format** can be downloaded here. The example file contains all the commits to the ArangoDB repository as shown by `git log --reverse`.

The following commands will import the data from the file into a collection named `commits`:

```
# download file
wget http://jsteemann.github.io/downloads/code/git-commits-single-line.json

# actually import data
arangoimp --file git-commits-single-line.json --collection commits --create-collection true
```

Note that no file type has been specified when `arangoimp` was invoked. This is because `json` is its default input format.

The other parameters used have the following meanings:

- `file` : input filename
- `collection` : name of the target collection
- `create-collection` : whether or not the collection should be created if it does not exist

The result of the import printed by `arangoimp` should be:

```
created:          20039
warnings/errors:  0
total:            20039
```

The collection `commits` should now contain the example commit data as present in the input file.

Importing JSON data in array format

An example input file for the **array format** can be found here.

The command for importing JSON data in **array format** is similar to what we've done before:

```
# download file
wget http://jsteemann.github.io/downloads/code/git-commits-array.json

# actually import data
arangoimp --file git-commits-array.json --collection commits --create-collection true
```

Though the import command is the same (except the filename), there is a notable difference between the two JSON formats: for the **array format**, `arangoimp` will read and parse the JSON in its entirety before it sends any data to the ArangoDB server. That means the whole input file must fit into `arangoimp` 's buffer. By default, `arangoimp` will allocate a 16 MiB internal buffer, and input files bigger than that will be rejected with the following message:

```
import file is too big. please increase the value of --batch-size (currently 16777216).
```

So for JSON input files in **array format** it might be necessary to increase the value of `--batch-size` in order to have the file imported. Alternatively, the input file can be converted to **line-by-line format** manually.

Importing CSV data

Data can also be imported from a CSV file. An example file can be found here.

The `--type` parameter for the import command must now be set to `csv` :

```
# download file
wget http://jsteemann.github.io/downloads/code/git-commits.csv

# actually import data
arangoimp --file git-commits.csv --type csv --collection commits --create-collection true
```

For the CSV import, the first line in the input file has a special meaning: every value listed in the first line will be treated as an attribute name for the values in all following lines. All following lines should also have the same number of "columns".

"columns" inside the CSV input file can be left empty though. If a "column" is left empty in a line, then this value will be omitted for the import so the respective attribute will not be set in the imported document. Note that values from the input file that are enclosed in double quotes will always be imported as strings. To import numeric values, boolean values or the `null` value, don't enclose these

values in quotes in the input file. Note that leading zeros in numeric values will be removed. Importing numbers with leading zeros will only work when putting the numbers into strings.

Here is an example CSV file:

```
"author","time","sha","message"
"Frank Celler","2011-10-26 08:42:49 +0200","c413859392a45873936cbe40797970f8eed93ff9","first commit"
"Frank Celler","2011-10-26 21:32:36 +0200","10bb77b8cc839201ff59a778f0c740994083c96e","initial release"
...
```

`arangoimp` supports Windows (CRLF) and Unix (LF) line breaks. Line breaks might also occur inside values that are enclosed with the quote character.

The default separator for CSV files is the comma. It can be changed using the `--separator` parameter when invoking `arangoimp`. The quote character defaults to the double quote ("). To use a literal double quote inside a "column" in the import data, use two double quotes. To change the quote character, use the `--quote` parameter. To use a backslash for escaping quote characters, please set the option `--backslash-escape` to `true`.

Changing the database and server endpoint

By default, `arangoimp` will connect to the default database on `127.0.0.1:8529` with a user named `root`. To change this, use the following parameters:

- `server.database`: name of the database to use when importing (default: `_system`)
- `server.endpoint`: address of the ArangoDB server (default: `tcp://127.0.0.1:8529`)

Using authentication

`arangoimp` will by default send an username `root` and an empty password to the ArangoDB server. This is ArangoDB's default configuration, and it should be changed. To make `arangoimp` use a different username or password, the following command-line arguments can be used:

- `server.username`: username, used if authentication is enabled on server
- `server.password`: password for user, used if authentication is enabled on server

The password argument can also be omitted in order to avoid having it saved in the shell's command-line history. When specifying a username but omitting the password parameter, `arangoimp` will prompt for a password.

Additional parameters

By default, `arangoimp` will import data into the specified collection but will not touch existing data. Often it is convenient to first remove all data from a collection and then run the import. `arangoimp` supports this with the optional `--overwrite` flag. When setting it to `true`, all documents in the collection will be removed prior to the import.

Author: Jan Steemann

Tags: #arangoimp #import

Replicating data from different databases

Problem

You have two or more different databases with various data respectively collections in each one of this, but you want your data to be collected at one place.

Note: For this solution you need at least Arango 2.0 and you must run the script in every database you want to be collect data from.

Solution

First of all you have to start a server on endpoint:

```
arangod --server.endpoint tcp://127.0.0.1:8529
```

Now you have to create two collections and name them *data* and *replicationStatus*

```
db._create("data");
db._create("replicationStatus");
```

Save the following script in a file named *js/common/modules/org/mysync.js*

```
var internal = require("internal");

// maximum number of changes that we can handle
var maxChanges = 1000;

// URL of central node
var transferUrl = "http://127.0.0.1:8599/_api/import?collection=central&type=auto&createCollection=true&complete=true";

var transferOptions = {
  method: "POST",
  timeout: 60
};

// the collection that keeps the status of what got replicated to central node
var replicationCollection = internal.db.replicationStatus;

// the collection containing all data changes
var changesCollection = internal.db.data;

function keyCompare (l, r) {
  if (l.length != r.length) {
    return l.length - r.length < 0 ? -1 : 1;
  }

  // length is equal
  for (i = 0; i < l.length; ++i) {
    if (l[i] != r[i]) {
      return l[i] < r[i] ? -1 : 1;
    }
  }

  return 0;
};

function logger (msg) {
  "use strict";

  require("console").log("%s", msg);
}

function replicate () {
  "use strict";
```

```
  var key = "status"; // const

  var status, newStatus;
  try {
    // fetch the previous replication state
    status = replicationCollection.document(key);
    newStatus = { _key: key, lastKey: status.lastKey };
  }
  catch (err) {
    // no previous replication state. start from the beginning
    newStatus = { _key: key, lastKey: "0" };
  }

  // fetch the latest changes (need to reverse them because `last` returns newest changes first)
  var changes = changesCollection.last(maxChanges).reverse(), change;
  var transfer = [ ];
  for (change in changes) {
    if (changes.hasOwnProperty(change)) {
      var doc = changes[change];
      if (keyCompare(doc._key, newStatus.lastKey) <= 0) {
        // already handled in a previous replication run
        continue;
      }

      // documents we need to transfer
      // if necessary, we could rewrite the documents here, e.g. insert
      // extra values, create client-specific keys etc.
      transfer.push(doc);

      if (keyCompare(doc._key, newStatus.lastKey) > 0) {
        // keep track of highest key
        newStatus.lastKey = doc._key;
      }
    }
  }

  if (transfer.length === 0) {
    // nothing to do
    logger("nothing to transfer");
    return;
  }

  logger("transferring " + transfer.length + " document(s)");

  // now transfer the documents to the remote server
  var result = internal.download(transferUrl, JSON.stringify(transfer), transferOptions);

  if (result.code >= 200 && result.code <= 202) {
    logger("central server accepted the documents: " + JSON.stringify(result));
  }
  else {
    // error
    logger("central server did not accept the documents: " + JSON.stringify(result));
    throw "replication error";
  }

  // update the replication state
  if (status) {
    // need to update the previous replication state
    replicationCollection.update(key, newStatus);
  }
  else {
    // need to insert the replication state (1st time)
    replicationCollection.save(newStatus);
  }

  logger("deleting old documents");

  // finally remove all elements that we transferred successfully from the changes collection
  // no need to keep them
  transfer.forEach(function (k) {
    changesCollection.remove(k);
  });
}

exports.execute = function (param) {
```

```
  "use strict";

  logger("replication wake up");
  replicate();
  logger("replication shutdown");
};
```

Afterwards change the URL of the central node in the script to the one you chosen before - e.g. *tcp://127.0.0.1:8599*

Now register the script as a recurring action:

```
require("internal").definePeriodic(1, 10, "org/arangodb/mysync", "execute", "");
```

Note: At this point you can change the time the script will be executed.

Comment

The server started on endpoint will be the central node. It collects changes from the local node by replicating its data. The script will pick up everything that has been changed since the last alteration in your *data* collection. Every 10 seconds - or the time you chosen - the script will be executed and send the changed data to the central node where it will be imported into a collection named *central*. After that the transferred data will be removed from the *data* collection.

If you want to test your script simply add some data to your *data* collection - e.g.:

```
for (i = 0; i < 100; ++i) db.data.save({ value: i });
```

Author: Jan Steemann

Tags: #database #collection

XCopy install ArangoDB on Windows

Problem

Even if there is a nice guided installer for windows users, not all users prefer this kind of installation. In order to have a portable application XCOPY deployment is necessary.

Solution

As of Version 2.5.1 ArangoDB doesn't rely on registry entries anymore so we can deploy using a ZIP-file.

Steps

Unzip archive

Open an explorer, choose a place where you want ArangoDB to be and unzip the files there. It will create its own toplevel directory with the version number in the string.

Alter configuration

Optional:

Edit `etc\arangodb3\arangod.conf` if the default values don't suit your needs like:

- the location of the database files
- ports to bind

and so on.

Create Runtime directories

`arangod` leans on the existence of some directories in the **var** subdirectory, so you should create them:

```
C:\Program Files\ArangoDB-3.1.11>mkdir var\lib\arangodb
C:\Program Files\ArangoDB-3.1.11>mkdir var\lib\arangodb-apps
```

Run arangod

To start the database simply run it:

```
C:\Program Files\ArangoDB-3.1.11>usr\bin\arangod
```

Now it takes a while to open all its databases, load system facilities, bootstrap the JavaScript environments and many more. Once it's ready the output is:

```
INFO ArangoDB (version 3.1.11 [windows]) is ready for business. Have fun!
```

Now you can open the administrative webinterface in your browser using http://127.0.0.1:8529/.

Installing as service

If you don't want to run `arangod` from a cmd-shell each time installing it as a system service is the right thing to do. This requires administrative privileges. You need to *Run as Administrator* the cmd-shell. First we need to grant the SYSTEM-user access to our database directory, since `arangod` is going to be running as that user:

```
C:\Program Files\ArangoDB-3.1.11>icacls var /grant SYSTEM:F /t
```

Next we can install the service itself:

```
C:\Program Files\ArangoDB-3.1.11>usr\bin\arangod --install-service
```

Now you will have a new entry in the **Services** dialog labeled **ArangoDB - the multi-purpose database**. You can start it there or just do it on the `commandline` using:

```
C:\Program Files\ArangoDB-3.1.11>NET START ArangoDB
```

It will take a similar amount of time to start from the `comandline` above till the service is up and running. Since you don't have any console to inspect the startup, messages of the severity FATAL & ERROR are also output into the windows eventlog, so in case of failure you can have a look at the **Eventlog** in the **Managementconsole**

Author: Wilfried Goesgens

Tags: #windows #install

Migration from ArangoDB 2.8 to 3.0

Problem

I want to use ArangoDB 3.0 from now on but I still have data in ArangoDB 2.8. I need to migrate my data. I am running an ArangoDB 3.0 cluster (and possibly a cluster with ArangoDB 2.8 as well).

Solution

The internal data format changed completely from ArangoDB 2.8 to 3.0, therefore you have to dump all data using `arangodump` and then restore it to the new ArangoDB instance using `arangorestore`.

General instructions for this procedure can be found in the manual. Here, we cover some additional details about the cluster case.

Dumping the data in ArangoDB 2.8

Basically, dumping the data works with the following command (use `arangodump` from your ArangoDB 2.8 distribution!):

```
arangodump --server.endpoint tcp://localhost:8530 --output-directory dump
```

or a variation of it, for details see the above mentioned manual page and this section. If your ArangoDB 2.8 instance is a cluster, simply use one of the coordinator endpoints as the above `--server.endpoint`.

Restoring the data in ArangoDB 3.0

The output consists of JSON files in the output directory, two for each collection, one for the structure and one for the data. The data format is 100% compatible with ArangoDB 3.0, except that ArangoDB 3.0 has an additional option in the structure files for synchronous replication, namely the attribute `replicationFactor`, which is used to specify, how many copies of the data for each shard are kept in the cluster.

Therefore, you can simply use this command (use the `arangorestore` from your ArangoDB 3.0 distribution!):

```
arangorestore --server.endpoint tcp://localhost:8530 --input-directory dump
```

to import your data into your new ArangoDB 3.0 instance. See this page for details on the available command line options. If your ArangoDB 3.0 instance is a cluster, then simply use one of the coordinators as `--server.endpoint`.

That is it, your data is migrated.

Controling the number of shards and the replication factor

This procedure works for all four combinations of single server and cluster for source and destination respectively. If the target is a single server all simply works.

So it remains to explain how one controls the number of shards and the replication factor if the destination is a cluster.

If the source was a cluster, `arangorestore` will use the same number of shards as before, if you do not tell it otherwise. Since ArangoDB 2.8 does not have synchronous replication, it does not produce dumps with the `replicationFactor` attribute, and so `arangorestore` will use replication factor 1 for all collections. If the source was a single server, the same will happen, additionally, `arangorestore` will always create collections with just a single shard.

There are essentially 3 ways to change this behaviour:

1. The first is to create the collections explicitly on the ArangoDB 3.0 cluster, and then set the `--create-collection false` flag. In this case you can control the number of shards and the replication factor for each collection individually when you create them.
2. The second is to use `arangorestore`'s options `--default-number-of-shards` and `--default-replication-factor` (this option was

introduced in Version 3.0.2) respectively to specify default values, which are taken if the dump files do not specify numbers. This means that all such restored collections will have the same number of shards and replication factor.

3. If you need more control you can simply edit the structure files in the dump. They are simply JSON files, you can even first use a JSON pretty printer to make editing easier. For the replication factor you simply have to add a `replicationFactor` attribute to the `parameters` subobject with a numerical value. For the number of shards, locate the `shards` subattribute of the `parameters` attribute and edit it, such that it has the right number of attributes. The actual names of the attributes as well as their values do not matter. Alternatively, add a `numberOfShards` attribute to the `parameters` subobject, this will override the `shards` attribute (this possibility was introduced in Version 3.0.2).

Note that you can remove individual collections from your dump by deleting their pair of structure and data file in the dump directory. In this way you can restore your data in several steps or even parallelise the restore operation by running multiple `arangorestore` processes concurrently on different dump directories. You should consider using different coordinators for the different `arangorestore` processes in this case.

All these possibilities together give you full control over the sharding layout of your data in the new ArangoDB 3.0 cluster.

Show grants function

Problem

I'm looking for user database grants

Solution

Create a global function in your *.arangosh.rc* file like this:

```
global.show_grants = function () {
        let stmt;
        stmt=db._createStatement({"query": "FOR u in _users RETURN {\"user\": u.user, \"databases\": u.databases}"});
        console.log(stmt.execute().toString());
};
```

Now when you enter in arangosh, you can call **show_grants()** function.

Function out example

```
[object ArangoQueryCursor, count: 3, hasMore: false]


[
  {
    "user" : "foo",
    "databases" : {
      "_system" : "rw",
      "bar" : "rw"
    }
  },
  {
    "user" : "foo2",
    "databases" : {
      "bar" : "rw"
    }
  },
  {
    "user" : "root",
    "databases" : {
      "*" : "rw"
    }
  }
]
```

Compiling ArangoDB

Problem

You want to modify sources or add your own changes to ArangoDB.

Solution

Arangodb, as many other opensource projects nowadays is standing on the shoulder of giants. This gives us a solid foundation to bring you a uniq feature set, but it introduces a lot of dependencies that need to be in place in order to compile arangodb.

Since build infrastructures are very different depending on the target OS, choose your target from the recepies below.

- Compile on Debian
- Compile on Windows
- Running Custom Build
  - Recompiling jemalloc
  - OpenSSL 1.1

Compiling on Debian

Problem

You want to compile and run the devel branch, for example to test a bug fix. In this example the system is Debian based.

Solution

This solution was made using a fresh Debian Testing machine on Amazon EC2. For completeness, the steps pertaining to AWS are also included in this recipe.

Launch the VM

Optional

Login to your AWS account and launch an instance of Debian Testing. I used an 'm3.xlarge' since that has a bunch of cores, more than enough memory, optimized network and the instance store is on SSDs which can be switched to provisioned IOPs.

The Current AMI ID's can be found in the Debian Wiki: https://wiki.debian.org/Cloud/AmazonEC2Image/Jessie

Upgrade to the very latest version

Optional

Once your EC2 instance is up, login ad `admin` and `sudo su` to become `root`.

First, we remove the backports and change the primary sources.list

```
rm -rf /etc/apt/sources.list.d
echo "deb     http://http.debian.net/debian testing main contrib"  > /etc/apt/sources.list
echo "deb-src http://http.debian.net/debian testing main contrib" >> /etc/apt/sources.list
```

Update and upgrade the system. Make sure you don't have any broken/unconfigured packages. Sometimes you need to run safe/full upgrade more than once. When you're done, reboot.

```
apt-get install aptitude
aptitude -y update
aptitude -y safe-upgrade
aptitude -y full-upgrade
reboot
```

Install build dependencies

Mandatory

Before you can build ArangoDB, you need a few packages pre-installed on your system.

Login again and install them.

```
sudo aptitude -y install git-core \
    build-essential \
    libssl-dev \
    libjemalloc-dev \
    cmake \
    python2.7 \
sudo aptitude -y install libldap2-dev # Enterprise version only
```

Download the Source

Download the latest source using ***git***:

```
unix> git clone git://github.com/arangodb/arangodb.git
```

This will automatically clone the **devel** branch.

Note: if you only plan to compile ArangoDB locally and do not want to modify or push any changes, you can speed up cloning substantially by using the *--single-branch* and *--depth* parameters for the clone command as follows:

```
unix> git clone --single-branch --depth 1 git://github.com/arangodb/arangodb.git
```

Setup

Switch into the ArangoDB directory

```
unix> cd arangodb
unix> mkdir build
unix> cd build
```

In order to generate the build environment please execute

```
unix> cmake ..
```

to setup the Makefiles. This will check the various system characteristics and installed libraries. If you installed the compiler in a non standard location, you may need to specify it:

```
cmake -DCMAKE_C_COMPILER=/opt/bin/gcc -DCMAKE_CXX_COMPILER=/opt/bin/g++ ..
```

If you compile on MacOS, you should add the following options to the cmake command:

```
cmake .. -DOPENSSL_ROOT_DIR=/usr/local/opt/openssl -DCMAKE_OSX_DEPLOYMENT_TARGET=10.11
```

If you also plan to make changes to the source code of ArangoDB, you may want to compile with the `Debug` build type:

```
cmake .. -DCMAKE_BUILD_TYPE=Debug
```

The `Debug` target enables additional sanity checks etc. which would slow down production binaries. If no build type is specified, ArangoDB will be compiled with build type `RelWithDebInfo`, which is a compromise between good performance and medium debugging experience.

Other options valuable for development:

```
-DUSE_MAINTAINER_MODE=On
```

Needed if you plan to make changes to AQL language (which is implemented using a lexer and parser files in `arangod/Aql/grammar.y` and `arangod/Aql/tokens.ll`) or if you want to enable runtime assertions. To use the maintainer mode, your system has to contain the tools FLEX and BISON.

```
-DUSE_BACKTRACE=On
```

Use this option if you want to have C++ stacktraces attached to your exceptions. This can be useful to more quickly locate the place where an exception or an assertion was thrown. Note that this option will slow down the produces binaries a bit and requires building with maintainer mode.

```
-DUSE_OPTIMIZE_FOR_ARCHITECTURE=On
```

This will optimize the binary for the target architecture, potentially enabling more compiler optimizations, but making the resulting binary less portable.

ArangoDB will then automatically use the configuration from file *etc/relative/arangod.conf*.

```
-DUSE_FAILURE_TESTS=On
```

This option activates additional code in the server that intentionally makes the server crash or misbehave (e.g. by pretending the system ran out of memory) when certain tests are run. This option is useful for writing tests.

```
-DUSE_JEMALLOC=Off
```

By default ArangoDB will be built with a bundled version of the JEMalloc allocator. This however will not work when using runtime analyzers such as ASAN or Valgrind. In order to use these tools for instrumenting an ArangoDB binary, JEMalloc must be turned off during compilation.

shared memory

Gyp is used as makefile generator by V8. Gyp requires shared memory to be available, which may not if you i.e. compile in a chroot. You can make it available like this:

```
none /opt/chroots/ubuntu_precise_x64/dev/shm tmpfs rw,nosuid,nodev,noexec 0 2
devpts    /opt/chroots/ubuntu_precise_x64/dev/pts    devpts    gid=5,mode=620    0  0
```

Compilation

Compile the programs (server, client, utilities) by executing

```
make
```

in the build subdirectory. This will compile ArangoDB and create the binary executable in file `build/bin/arangod`.

Starting and testing

Check the binary by starting it using the command line.

```
unix> build/bin/arangod -c etc/relative/arangod.conf --server.endpoint tcp://127.0.0.1:8529 /tmp/database-dir
```

This will start up the ArangoDB and listen for HTTP requests on port 8529 bound to IP address 127.0.0.1. You should see the startup messages similar to the following:

```
2016-06-01T12:47:29Z [29266] INFO ArangoDB xxx ...
2016-06-10T12:47:29Z [29266] INFO using endpoint 'tcp://127.0.0.1.8529' for non-encrypted requests
2016-06-01T12:47:30Z [29266] INFO Authentication is turned on
2016-60-01T12:47:30Z [29266] INFO ArangoDB (version xxx) is ready for business. Have fun!
```

If it fails with a message about the database directory, please make sure the database directory you specified exists and can be written into.

Use your favorite browser to access the URL

```
http://127.0.0.1:8529/
```

This should bring up ArangoDB's web interface.

Re-building ArangoDB after an update

To stay up-to-date with changes made in the main ArangoDB repository, you will need to pull the changes from it and re-run `make`.

Normally, this will be as simple as follows:

```
unix> git pull
unix> (cd build && make)
```

From time to time there will be bigger structural changes in ArangoDB, which may render the old Makefiles invalid. Should this be the case and `make` complains about missing files etc., the following commands should fix it:

```
unix> rm -rf build/*
unix> cd build && cmake .. <cmake options go here>
unix> (cd build && make)
```

Note that the above commands will run a full rebuild of ArangoDB and all of its third-party components. That will take a while to complete.

Installation

In a local development environment it is not necessary to install ArangoDB somewhere, because it can be started from within the source directory as shown above.

If there should be the need to install ArangoDB, execute the following command:

```
(cd build && sudo make install)
```

The server will by default be installed in

```
/usr/local/sbin/arangod
```

The configuration file will be installed in

```
/usr/local/etc/arangodb/arangod.conf
```

The database will be installed in

```
/usr/local/var/lib/arangodb
```

The ArangoShell will be installed in

```
/usr/local/bin/arangosh
```

You should add an arangodb user and group (as root), plus make sure it owns these directories:

```
useradd -g arangodb arangodb
chown -R arangodb:arangodb /usr/local/var/lib/arangodb3-apps/
chown -R arangodb:arangodb /tmp/database-dir/
```

Note: The installation directory will be different if you use one of the `precompiled` packages. Please check the default locations of your operating system, e. g. `/etc` and `/var/lib` .

When upgrading from a previous version of ArangoDB, please make sure you inspect ArangoDB's log file after an upgrade. It may also be necessary to start ArangoDB with the *--database.auto-upgrade* parameter once to perform required upgrade or initialization tasks.

Author: Patrick Huber **Author:** Wilfried Goesgens

Tags: #debian #driver

Compiling ArangoDB under Windows

Problem

I want to compile ArangoDB 3.0 and onwards under Windows.

Note: For this recipe you need at least ArangoDB 3.0; For ArangoDB version before 3.0 look at the old Compiling ArangoDB under Windows.

Solution

With ArangoDB 3.0 a complete cmake environment was introduced. This also streamlines the dependencies on windows. We sugest to use chocolatey.org to install most of the dependencies. For sure most projects offer their own setup & install packages, chocolatey offers a simplified way to install them with less userinteractions. You can even use chocolatey via the brand new ansibles 2.0 winrm facility to do unattended installions of some software on windows - the cool thing linux guys always told you about.

Ingredients

First install the choco package manager by pasting this tiny cmdlet into a command window *(needs to be run with Administrator privileges; Right click start menu, **Command Prompt (Admin)**)*:

```
@powershell -NoProfile -ExecutionPolicy Bypass -Command "iex ((new-object net.webclient).DownloadString('https://chocolatey.org
/install.ps1'))" && SET PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin
```

Visual Studio and its Compiler

Since choco currently fails to alter the environment for Microsoft Visual Studio, we suggest to download and install Visual Studio by hand. Currently Visual Studio 2015 is the only supported option.

You need to make sure that it installs the option "Programming Languages / C++", else cmake will fail to dectect it later on.

After it successfully installed, start it once, so it can finish its setup.

More dependencies

Now you can invoke the choco package manager for an unattended install of the dependencies *(needs to be run with Administrator privileges again)*:

```
choco install -y cmake.portable nsis python2 procdump windbg wget nuget.commandline
```

Then we fetch the OpenSSL library via the nuget commandline client *(doesn't need Administrator privileges)*:

```
nuget install openssl
```

Optional

If you intend to run the unittests or compile from git, you also need *(needs to be run with Administrator privileges again)*:

```
choco install -y git winflexbison ruby
```

Close and reopen the Administrator command window in order to continue with the ruby devkit:

```
choco install -y ruby2.devkit
```

And manually install the requirements via the `Gemfile` fetched from the ArangoDB Git repository *(needs to be run with Administrator privileges)*:

```
wget https://raw.githubusercontent.com/arangodb/arangodb/devel/UnitTests/HttpInterface/Gemfile
set PATH=%PATH%;C:\tools\DevKit2\bin;C:\tools\DevKit2\mingw\bin
gem install bundler
bundler
```

Note that the V8 build scripts and gyp aren't compatible with Python 3.x hence you need python2!

Building ArangoDB

Download and extract the release tarball from https://www.arangodb.com/download/

Or clone the github repository, and checkout the branch or tag you need (devel, 3.0)

```
git clone https://github.com/arangodb/arangodb.git -b devel
cd arangodb
```

Generate the Visual studio project files, and check back that cmake discovered all components on your system:

```
mkdir Build64
cd Build64
cmake -G "Visual Studio 14 Win64" ..
```

Note that in some cases cmake struggles to find the proper python interpreter (i.e. the cygwin one won't work). You can force overrule it by appending:

```
-DPYTHON_EXECUTABLE:FILEPATH=C:/tools/python2/python.exe
```

You can now load these in the Visual Studio IDE or use cmake to start the build:

```
cmake --build . --config RelWithDebInfo
```

The binaries need the ICU datafile `icudt54l.dat`, which is automatically copied into the directory containing the executable.

For development, unittests and documentation: Cygwin (Optional)

The documentation and unittests still require a cygwin environment. Here the hints how to get it properly installed:

You need at least `make` from cygwin. Cygwin also offers a `cmake`. Do **not** install the cygwin cmake.

You should also issue these commands to generate user informations for the cygwin commands:

```
mkpasswd > /etc/passwd
mkgroup > /etc/group
```

Turning ACL off (noacl) for all mounts in cygwin fixes permissions troubles that may appear in the build:

```
# /etc/fstab
#
#    This file is read once by the first process in a Cygwin process tree.
#    To pick up changes, restart all Cygwin processes.  For a description
#    see https://cygwin.com/cygwin-ug-net/using.html#mount-table

# noacl = Ignore Access Control List and let Windows handle permissions
C:/cygwin64/bin  /usr/bin   ntfs      binary,auto,noacl           0  0
C:/cygwin64/lib  /usr/lib   ntfs      binary,auto,noacl           0  0
C:/cygwin64      /          ntfs      override,binary,auto,noacl  0  0
```

```
none            /cygdrive  cygdrive  binary,posix=0,user,noacl   0  0
```

Enable native symlinks for Cygwin and git

Cygwin will create proprietary files as placeholders by default instead of actually symlinking files. The placeholders later tell Cygwin where to resolve paths to. It does not intercept every access to the placeholders however, so that 3rd party scripts break. Windows Vista and above support real symlinks, and Cygwin can be configured to make use of it:

```
# use actual symlinks to prevent documentation build errors
# (requires elevated rights!)
export CYGWIN="winsymlinks:native"
```

Note that you must run Cygwin as administrator or change the Windows group policies to allow user accounts to create symlinks (`gpedit.msc` if available).

BTW: You can create symlinks manually on Windows like:

```
mklink /H target/file.ext source/file.ext
mklink /D target/path source/path
mklink /J target/path source/path/for/junction
```

And in Cygwin:

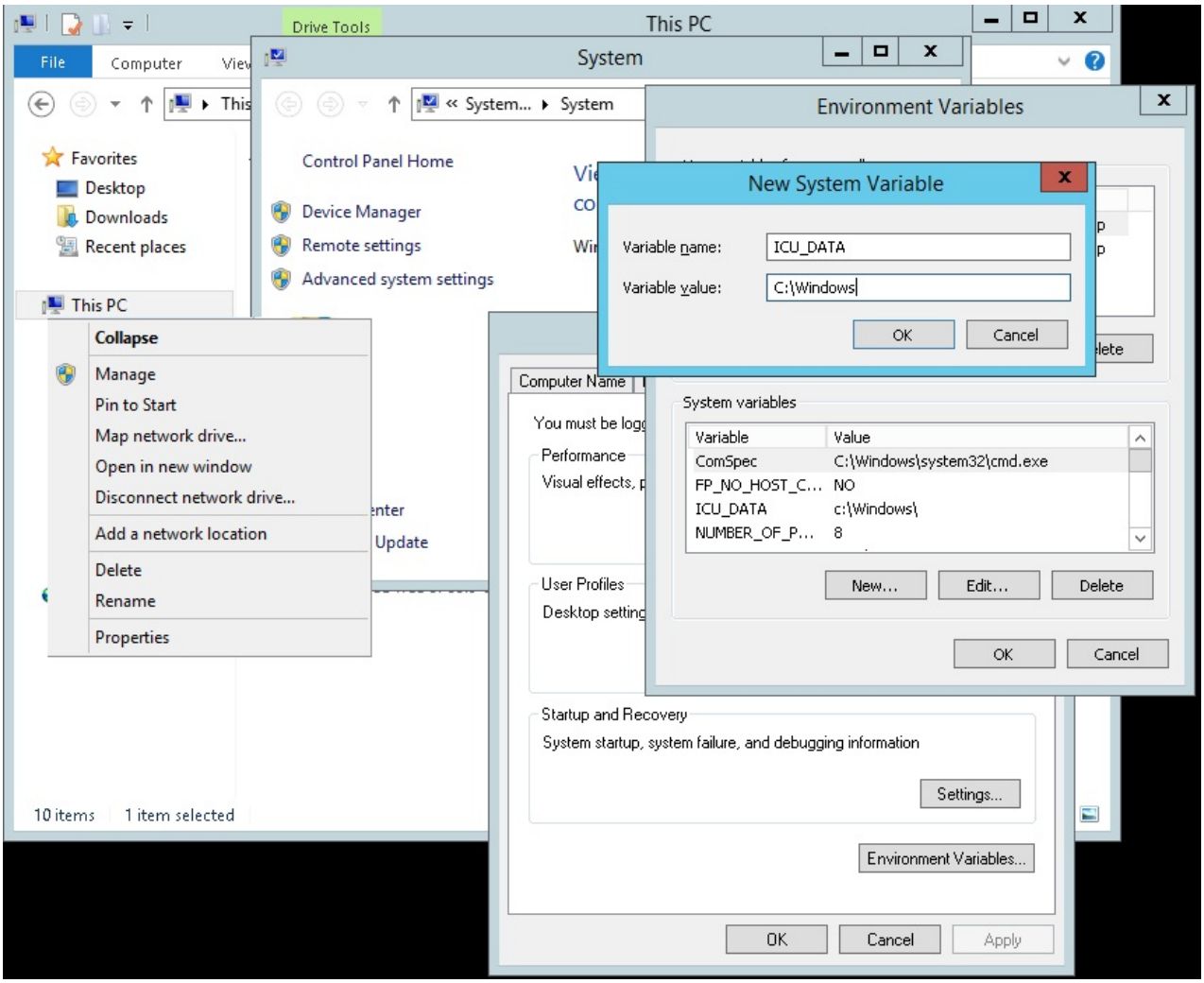
```
ln -s source target
```

Making the ICU database publically available

If you intend to use the machine for development purposes, it may be more practical to copy it to a common place:

```
cp 3rdParty/V8/V8-5.0.71.39/third_party/icu/source/data/in/icudtl.dat /cygdrive/c/Windows/icudt54l.dat
```

And configure your environment (yes this instruction remembers to the hitchhikers guide to the galaxy...) so that `ICU_DATA` points to `c:\\Windows`. You do that by opening the explorer, right click on `This PC` in the tree on the left, choose `Properties` in the opening window `Advanced system settings`, in the Popup `Environment Variables`, another popup opens, in the `System Variables` part you click `New`, And variable name: `ICU_DATA` to the value: `c:\\Windows`



Running Unitests (Optional)

You can then run the unittests in the cygwin shell like that:

```
build64/bin/RelWithDebInfo/arangosh.exe \
-c etc/relative/arangosh.conf \
--log.level warning \
--server.endpoint tcp://127.0.0.1:1024 \
--javascript.execute UnitTests/unittest.js \
  -- \
  all \
  --ruby c:/tools/ruby22/bin/ruby \
  --rspec c:/tools/ruby22/bin/rspec \
  --buildType RelWithDebInfo \
  --skipNondeterministic true \
  --skipTimeCritical true
  --skipBoost true \
  --skipGeo true
```

Documentation (Optional)

NodeJS *(needs to be run with Administrator privileges again)*:

```
choco install -y nodejs
```

Gitbook:

```
npm install -g gitbook-cli
```

Markdown-pp:

```
git clone https://github.com/triAGENS/markdown-pp.git
cd markdown-pp
python setup.py install
```

Ditaa:

```
Download and install: http://ditaa.sourceforge.net/#download
```

Authors: Frank Celler, Wilfried Goesgens and Simran Brucherseifer.

Tags: #windows

OpenSSL

OpenSSL 1.1 is on its way to mainstream. So far (ArangoDB 3.2) has only been thorougly tested with OpenSSL 1.0 and 1.1 is unsupported.

Building against 1.1 will currently result in a compile error:

```
/arangodb/arangodb/lib/SimpleHttpClient/SslClientConnection.cpp:224:14: error: use of undeclared identifier 'SSLv2_method'
      meth = SSLv2_method();
             ^
/arangodb/arangodb/lib/SimpleHttpClient/SslClientConnection.cpp:239:14: warning: 'TLSv1_method' is deprecated [-Wdeprecated-dec
larations]
      meth = TLSv1_method();
             ^
/usr/include/openssl/ssl.h:1612:45: note: 'TLSv1_method' has been explicitly marked deprecated here
DEPRECATEDIN_1_1_0(__owur const SSL_METHOD *TLSv1_method(void)) /* TLSv1.0 */
                                            ^
/arangodb/arangodb/lib/SimpleHttpClient/SslClientConnection.cpp:243:14: warning: 'TLSv1_2_method' is deprecated [-Wdeprecated-d
eclarations]
      meth = TLSv1_2_method();
```

You should install openssl 1.0 (should be possible to install it alongside 1.1).

After that help cmake to find the 1.0 variant.

Example on Arch Linux:

```
cmake -DOPENSSL_INCLUDE_DIR=/usr/include/openssl-1.0/ -DOPENSSL_SSL_LIBRARY=/usr/lib/libssl.so.1.0.0 -DOPENSSL_CRYPTO_LIBRARY=/
usr/lib/libcrypto.so.1.0.0 <SOURCE_PATH>
```

After that ArangoDB should compile fine.

Running a custom build

Problem

You've already built a custom version of ArangoDB and want to run it. Possibly in isolation from an existing installation or you may want to re-use the data.

Solution

First, you need to build your own version of ArangoDB. If you haven't done so already, have a look at any of the Compiling recipes.

This recipe assumes you're in the root directory of the ArangoDB distribution and compiling has successfully finished.

Running in isolation

This part shows how to run your custom build with an empty database directory

```
# create data directory
mkdir /tmp/arangodb

# run
bin/arangod \
    --configuration etc/relative/arangod.conf\
     --database.directory /tmp/arangodb
```

Running with data

This part shows how to run your custom build with the config and data from a pre-existing stable installation.

BEWARE ArangoDB's developers may change the db file format and after running with a changed file format, there may be no way back. Alternatively you can run your build in isolation and dump and restore the data from the stable to your custom build.

When running like this, you must run the db as the arangod user (the default installed by the package) in order to have write access to the log, database directory etc. Running as root will likely mess up the file permissions - good luck fixing that!

```
# become root first
su

# now switch to arangod and run
su - arangod
bin/arangod --configuration /etc/arangodb/arangod.conf
```

Author: Patrick Huber

Tags: #build

Jemalloc

This article is only relevant if you intend to compile arangodb on Ubuntu 16.10 or debian testing

On more modern linux systems (development/floating at the time of this writing) you may get compile / link errors with arangodb regarding jemalloc. This is due to compilers switching their default behaviour regarding the `PIC` - Position Independend Code. It seems common that jemalloc remains in a stage where this change isn't followed and causes arangodb to error out during the linking phase.

From now on cmake will detect this and give you this hint:

```
the static system jemalloc isn't suitable! Recompile with the current compiler or disable using `-DCMAKE_CXX_FLAGS=-no-pie -DCMAKE_C_FLAGS=-no-pie`
```

Now you've got three choices.

Doing without jemalloc

Fixes the compilation issue, but you will get problems with the glibcs heap fragmentation behaviour which in the longer run will lead to an ever increasing memory consumption of ArangoDB.

So, while this may be suitable for development / testing systems, its definitely not for production.

Disabling PIC altogether

This will build an arangod which doesn't use this compiler feature. It may be not so nice for development builds. It can be achieved by specifying these options on cmake:

```
-DCMAKE_CXX_FLAGS=-no-pie -DCMAKE_C_FLAGS=-no-pie
```

Recompile jemalloc

The smartest way is to fix the jemalloc libraries packages on your system so its reflecting that new behaviour. On debian / ubuntu systems it can be achieved like this:

```
apt-get install automake debhelper docbook-xsl xsltproc dpkg-dev
apt source jemalloc
cd jemalloc*
dpkg-buildpackage
cd ..
dpkg -i *jemalloc*deb
```

Cloud, DCOS and Docker

Amazon Web Services (AWS)

- Running on AWS
- Update on AWS

Microsoft Azure

- Running on Azure

Docker

- Docker ArangoDB
- Docker with NodeJS App

GiantSwarm

- In the GiantSwarm

Mesos / DCOS

- ArangoDB in Mesos
- DC/OS: Full example

Running ArangoDB on AWS

ArangoDB is available as AMI on the AWS Marketplace.

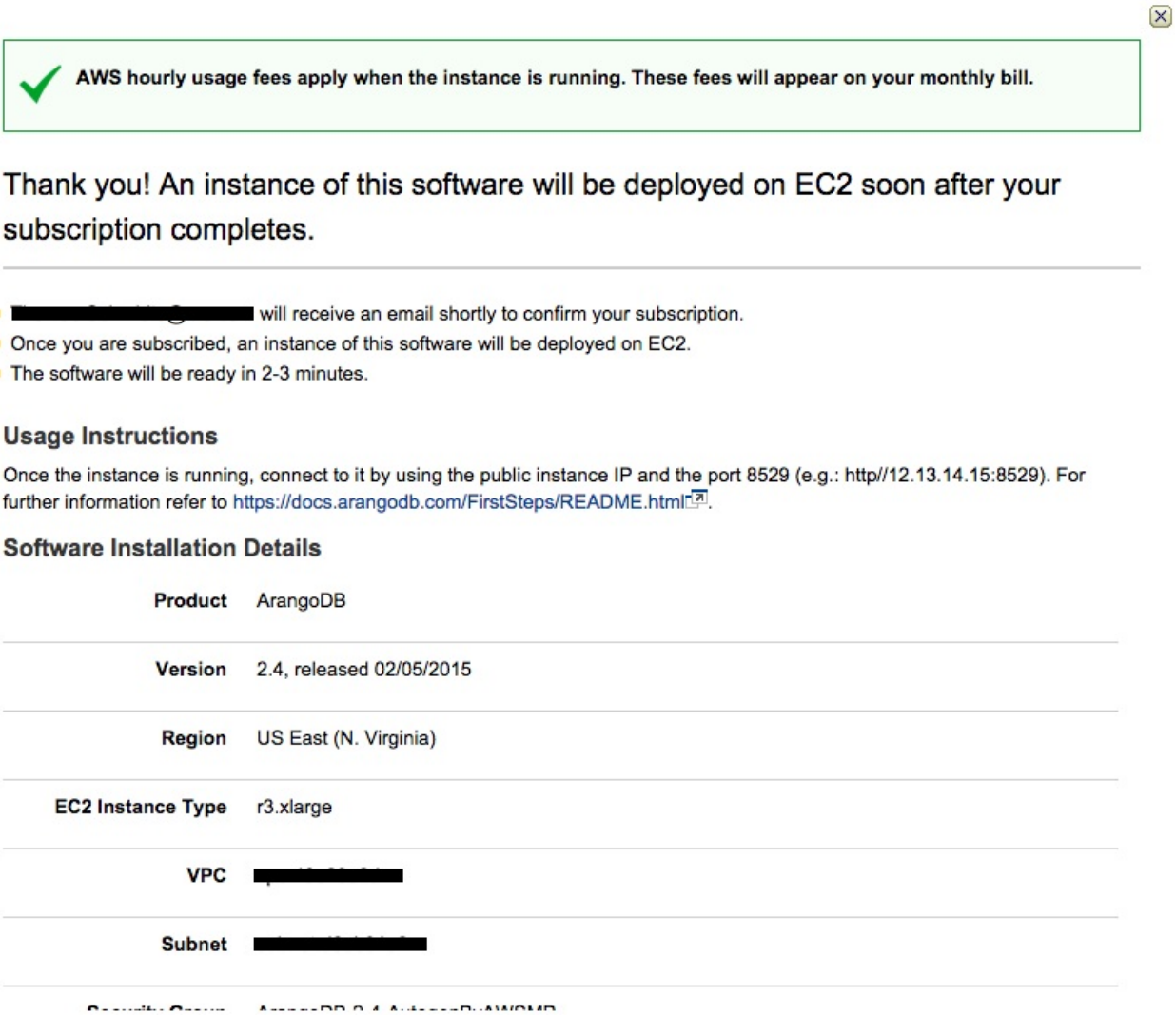
(If you've already a running ArangoDB image on AWS and need an update, please have a look at Updating ArangoDB on AWS).

Here is a quick guide how to start:

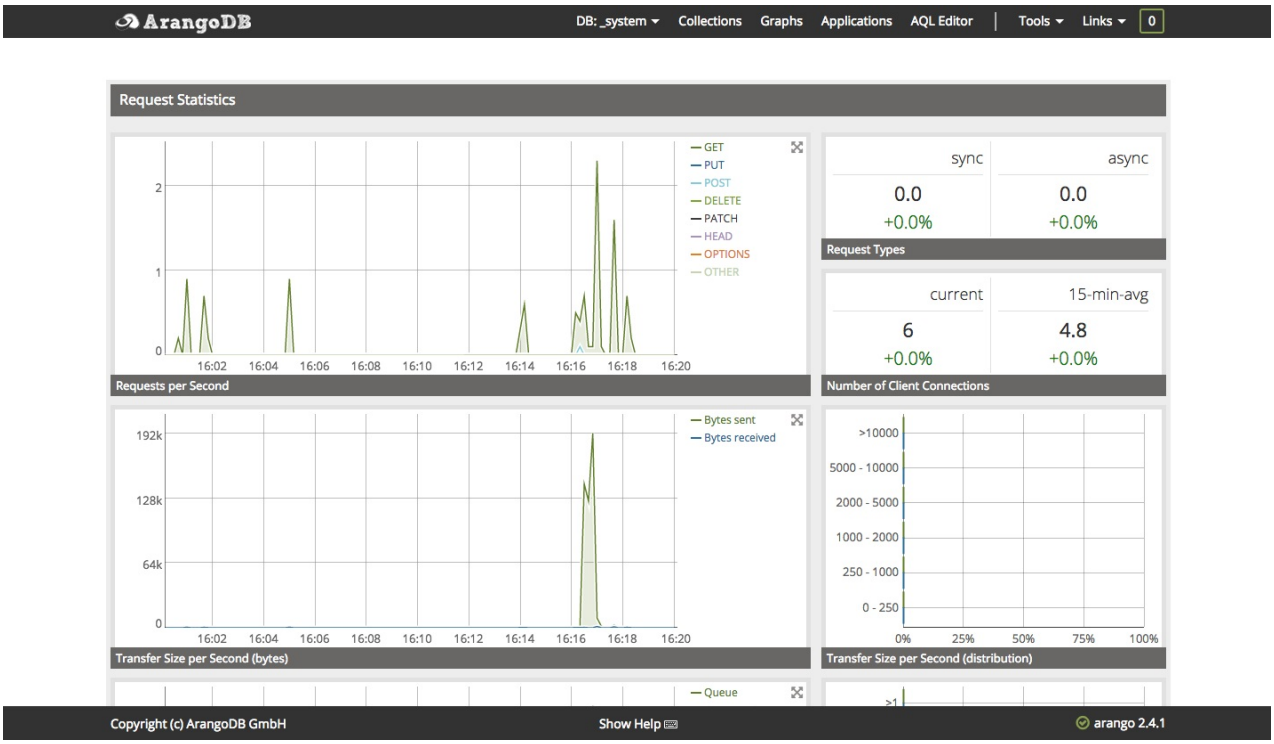
- Go the ArangoDB marketplace, select the latest version and click on **Continue**
- Use the **1-Click Launch** tab and select the size of the instance (**EC2 Instance Type**) you wish to use.
- Now you can continue with a click on **Accept Terms & Launch with 1-Click**.

Note: If you do not have a key pair a warning will appear after clicking and you will be asked to generate a key pair.

You successfully launched an ArangoDB instance on AWS.



The ArangoDB Web-Interface can be reached using the **Access Software** button or via public instance IP and the Port 8529 (e.g.: http://12.13.14.15:8529) The default user is `root` and the password is the `Instance ID` (You can find the Instance ID on the instance list).



If you want to learn more about ArangoDB, start with the [ArangoDB First Steps][../../Manual/GettingStarted/index.html] in our Documentation or try one of our Tutorials or Cookbook recipes.

Author: Ingo Friepoertner

Tags : #aws, #amazon, #howto

Updating an ArangoDB Image on AWS

If you run an ArangoDB on AWS and used the provided AMI in the AWS Marketplace, you at some point want to update to the latest release. The process to submit and publish a new ArangoDB image to the marketplace takes some time and you might not find the latest release in the marketplace store yet.

However, updating to the latest version is not that hard.

First, log in to the virtual machine with the user *ubuntu* and the public DNS name of the instance.

```
ssh ubuntu@ec2-XX-XX-XXX-XX.us-west-2.compute.amazonaws.com
```

To start an update to a known version of ArangoDB you can use:

```
sudo apt-get update
sudo apt-get install arangodb=2.5.7
```

To upgrade an ArangoDB instance to a new major version (from 2.5.x to 2.6.x), use:

```
sudo apt-get install arangodb
```

You might get a warning that the configuration file has changed:

```
Configuration file '/etc/arangodb/arangod.conf'
==> Modified (by you or by a script) since installation.
==> Package distributor has shipped an updated version.
  What would you like to do about it ?  Your options are:
   Y or I  : install the package maintainer's version
   N or O  : keep your currently-installed version
     D     : show the differences between the versions
     Z     : start a shell to examine the situation
The default action is to keep your current version.
*** arangod.conf (Y/I/N/O/D/Z) [default=N] ?
```

You should stay with the current configuration (type "N"), as there are some changes made in the configuration for AWS. If you type "Y" you will lose access from your applications to the database so make sure that database directory and server endpoint are valid.

```
 --server.database-directory
    needs to be  `/vol/...` for AWS
 --server.endpoint
    needs to be `tcp://0.0.0.0:8529` for AWS
```

If you update to a new major version, you will be asked to `upgrade` so that a database migration can be started:

```
sudo service arangodb upgrade
sudo service arangodb start
```

Now ArangoDB should be back to normal.

For now we have to stick with this manual process but e might create a simpler update process in the future. Please provide feedback how you use our Amazon AMI and how we can improve your user experience.

Author: Ingo Friepoertner

Tags: #aws #upgrade

ArangoDB in Microsoft Azure

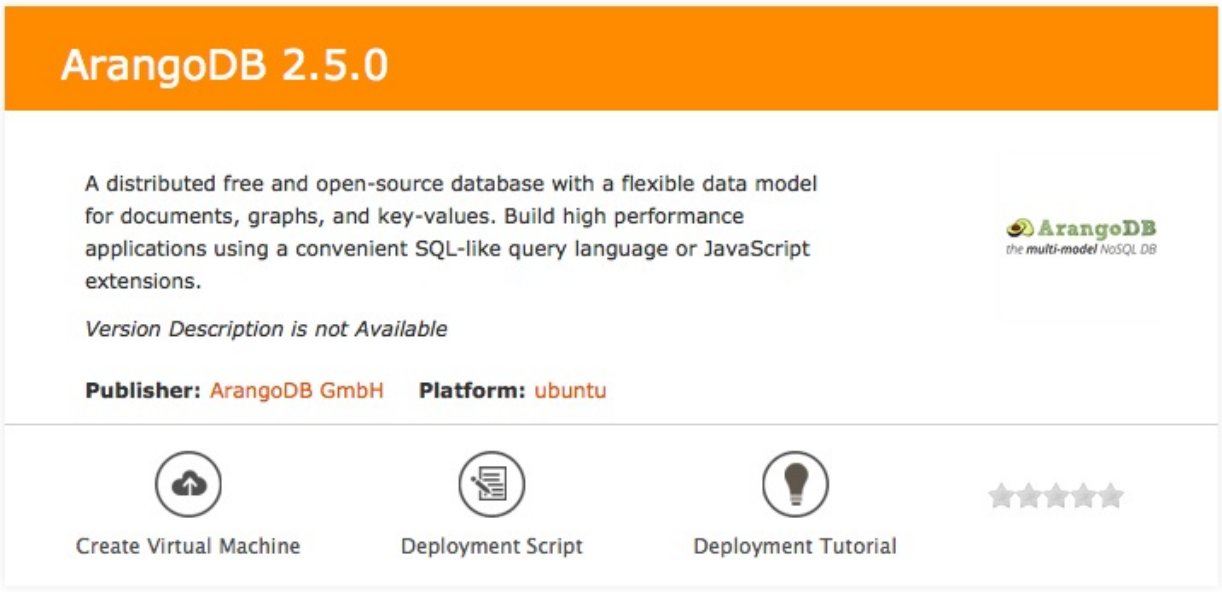
I want to use ArangoDB in Microsoft Azure

How to

The short answer is: go to

```
https://vmdepot.msopentech.com/
```

type in "ArangoDB", select the version you require and press "Create Virtual Machine".



1

Follow the instructions given there and within minutes you have a running ArangoDB instance in Microsoft Azure. You will receive an email as soon as your machine is ready.

Assume your machine is called `myarangodb`, than you can access ArangoDB pointing your browser to

```
http://myarangodb.cloudapp.net:8529
```

Please note that for security reasons the default instance is password protected.

However, the password for "root" is empty. So, please log in and change the password as soon as possible.

Authors: Frank Celler

Tags: #azure, #howto

How to run ArangoDB in a Docker container

Problem

How do you make ArangoDB run in a Docker container?

Solution

ArangoDB is now available as an official repository in the Docker Hub (@see documentation there).

Author: Frank Celler

Tags: #docker #howto

ArangoDB, NodeJS and Docker

Problem

I'm looking for a head start in using the ArangoDB docker image.

Solution

We will use the guesser game for ArangoDB from

```
https://github.com/arangodb/guesser
```

This is a simple game guessing animals or things. It learns while playing and stores the learned information in an ArangoDB instance. The game is written using the express framework.

Note: You need to switch to the docker branch.

The game has the two components

- front-end with node.js and express
- back-end with ArangoDB and Foxx

Therefore the guesser game needs two docker containers, one container for the node.js server to run the front-end code and one container for ArangoDB for the storage back-end.

Node Server

The game is itself can be install via NPM or from github. There is an image available from dockerhub called `arangodb/example-guesser` which is based on the Dockerfile from github.

You can either build the docker container locally or simply use the available one from docker hub.

```
unix> docker run -p 8000:8000 -e nolink=1 arangodb/example-guesser
Starting without a database link
Using DB-Server http://localhost:8529
Guesser app server listening at http://0.0.0.0:8000
```

This will start-up node and the guesser game is available on port 8000. Now point your browser to port 8000. You should see the start-up screen. However, without a storage backend it will be pretty useless. Therefore, stop the container and proceed with the next step.

If you want to build the container locally, check out the guesser game from

```
https://github.com/arangodb/example-guesser
```

Switch into the `docker/node` subdirectory and execute `docker build .`.

ArangoDB

ArangoDB is already available on docker, so we start an instance

```
unix> docker run --name arangodb-guesser arangodb/arangodb
show all options:
  docker run -e help=1 arangodb

starting ArangoDB in stand-alone mode
```

That's it. Note that in an productive environment you would need to attach a storage container to it. We ignore this here for the sake of simplicity.

Guesser Game

Some Testing

Use the guesser game image to start the ArangoDB shell and link the ArangoDB instance to it.

```
unix> docker run --link arangodb-guesser:db-link -it arangodb/example-guesser arangosh --server.endpoint @DB_LINK_PORT_8529_TCP
@
```

The parameter `--link arangodb-guesser:db-link` links the running ArangoDB into the application container and sets an environment variable `DB_LINK_PORT_8529_TCP` which points to the exposed port of the ArangoDB container:

```
DB_LINK_PORT_8529_TCP=tcp://172.17.0.17:8529
```

Your IP may vary. The command `arangosh ...` at the end of docker command executes the ArangoDB shell instead of the default node command.

```
Welcome to arangosh 2.3.1 [linux]. Copyright (c) ArangoDB GmbH
Using Google V8 3.16.14 JavaScript engine, READLINE 6.3, ICU 52.1

Pretty printing values.
Connected to ArangoDB 'tcp://172.17.0.17:8529' version: 2.3.1, database: '_system', username: 'root'

Type 'tutorial' for a tutorial or 'help' to see common examples
arangosh [_system]>
```

The important line is

```
Connected to ArangoDB 'tcp://172.17.0.17:8529' version: 2.3.1, database: '_system', username: 'root'
```

It tells you that the application container was able to connect to the database back-end. Press `Control-D` to exit.

Start Up The Game

Ready to play? Start the front-end container with the database link and initialize the database.

```
unix> docker run --link arangodb-guesser:db-link -p 8000:8000 -e init=1 arangodb/example-guesser
```

Use your browser to play the game at the address http://127.0.0.1:8000/. The

```
-e init=1
```

is only need the first time you start-up the front-end and only once. The next time you run the front-end or if you start a second front-end server use

```
unix> docker run --link arangodb-guesser:db-link -p 8000:8000 arangodb/example-guesser
```

Author: Frank Celler

Tags: #docker

ArangoDB in the Giant Swarm using Docker containers

Problem

I want to use ArangoDB in the Giant Swarm with Docker containers.

Solution

Giant Swarm allows you to describe and deploy your application by providing a simple JSON description. The current weather app is a good example on how to install an application which uses two components, namely `node` and `redis`.

My colleague Max has written a guesser game with various front-ends and ArangoDB as backend. In order to get the feeling of being part of the Giant Swarm, I have started to set up this game in the swarm.

First Steps

The guesser game consists of a front-end written as express application in node and a storage back-end using ArangoDB and a small API developed with Foxx.

The front-end application is available as image

```
arangodb/example-guesser
```

and the ArangoDB back-end with the Foxx API as

```
arangodb/example-guesser-db
```

The dockerfiles used to create the images are available from github

```
https://github.com/arangodb/guesser
```

Set up the Swarm

Set up your swarm environment as described in the documentation. Create a configuration file for the swarm called `arangodb.json` and fire up the application

```
{
  "app_name": "guesser",
  "services": [
    {
      "service_name": "guesser-game",
      "components": [
        {
          "component_name": "guesser-front-end",
          "image": "arangodb/example-guesser",
          "ports": [ 8000 ],
          "dependencies": [
            { "name": "guesser-back-end", "port": 8529 }
          ],
          "domains": { "guesser.gigantic.io": 8000 }
        },
        {
          "component_name": "guesser-back-end",
          "image": "arangodb/example-guesser-db",
          "ports": [ 8529 ]
        }
      ]
    }
  ]
}
```

This defines an application `guesser` with a single service `guesser-game`. This service has two components `guesser-front-end` and `guesser-back-end`. The docker images are downloaded from the standard docker repository.

The line

```
"domains": { "guesser.gigantic.io": 8000 }
```

exposes the internal port 8000 to the external port on port 80 for the host `guesser.gigantic.io`.

In order to tell Giant Swarm about your application, execute

```
unix> swarm create arangodb.json
Creating 'arangodb' in the 'fceller/dev' environment...
App created successfully!
```

This will create an application called `guesser`.

```
unix> swarm status guesser
App guesser is down

service       component          instanceid                           status
guesser-game  guesser-back-end   5347e718-3d27-4356-b530-b24fc5d1e3f5  down
guesser-game  guesser-front-end  7cf25b43-13c4-4dd3-9a2b-a1e32c43ae0d  down
```

We see the two components of our application. Both are currently powered down.

Startup the Guesser Game

Starting your engines is now one simple command

```
unix> swarm start guesser
Starting application guesser...
Application guesser is up
```

Now the application is up

```
unix> swarm status guesser
App guesser is up

service       component          instanceid                           status
guesser-game  guesser-back-end   5347e718-3d27-4356-b530-b24fc5d1e3f5  up
guesser-game  guesser-front-end  7cf25b43-13c4-4dd3-9a2b-a1e32c43ae0d  up
```

Point your browser to

```
http://guesser.gigantic.io
```

and guess an animal.

If you want to check the log files of an instance you can ask the swarm giving it the instance id. For example, the back-end

```
unix> swarm logs 5347e718-3d27-4356-b530-b24fc5d1e3f5
2014-12-17 12:34:57.984554 +0000 UTC - systemd - Stopping User guesser-back-end...
2014-12-17 12:36:28.074673 +0000 UTC - systemd - 5cfe11d6-343e-49bb-8029-06333844401f.service stop-sigterm timed out. Killing.
2014-12-17 12:36:28.077821 +0000 UTC - systemd - 5cfe11d6-343e-49bb-8029-06333844401f.service: main process exited, code=killed
, status=9/KILL
2014-12-17 12:36:38.213245 +0000 UTC - systemd - Stopped User guesser-back-end.
2014-12-17 12:36:38.213543 +0000 UTC - systemd - Unit 5cfe11d6-343e-49bb-8029-06333844401f.service entered failed state.
2014-12-17 12:37:55.074158 +0000 UTC - systemd - Starting User guesser-back-end...
2014-12-17 12:37:55.208354 +0000 UTC - docker  - Pulling repository arangodb/example-guesser-db
2014-12-17 12:37:56.995122 +0000 UTC - docker  - Status: Image is up to date for arangodb/example-guesser-db:latest
2014-12-17 12:37:57.000922 +0000 UTC - systemd - Started User guesser-back-end.
2014-12-17 12:37:57.707575 +0000 UTC - docker  - --> starting ArangoDB
```

```
2014-12-17 12:37:57.708182 +0000 UTC - docker  - --> waiting for ArangoDB to become ready
2014-12-17 12:38:28.157338 +0000 UTC - docker  - --> installing guesser game
2014-12-17 12:38:28.59025 +0000 UTC  - docker  - --> ready for business
```

and the front-end

```
unix> swarm logs 7cf25b43-13c4-4dd3-9a2b-a1e32c43ae0d
2014-12-17 12:35:10.139684 +0000 UTC - systemd - Stopping User guesser-front-end...
2014-12-17 12:36:40.32462 +0000 UTC  - systemd - aa7756a4-7a87-4633-bea3-e416d035188b.service stop-sigterm timed out. Killing.
2014-12-17 12:36:40.327754 +0000 UTC - systemd - aa7756a4-7a87-4633-bea3-e416d035188b.service: main process exited, code=killed
, status=9/KILL
2014-12-17 12:36:50.567911 +0000 UTC - systemd - Stopped User guesser-front-end.
2014-12-17 12:36:50.568204 +0000 UTC - systemd - Unit aa7756a4-7a87-4633-bea3-e416d035188b.service entered failed state.
2014-12-17 12:38:04.796129 +0000 UTC - systemd - Starting User guesser-front-end...
2014-12-17 12:38:04.921273 +0000 UTC - docker  - Pulling repository arangodb/example-guesser
2014-12-17 12:38:06.459366 +0000 UTC - docker  - Status: Image is up to date for arangodb/example-guesser:latest
2014-12-17 12:38:06.469988 +0000 UTC - systemd - Started User guesser-front-end.
2014-12-17 12:38:07.391149 +0000 UTC - docker  - Using DB-Server http://172.17.0.183:8529
2014-12-17 12:38:07.613982 +0000 UTC - docker  - Guesser app server listening at http://0.0.0.0:8000
```

Scaling Up

Your game becomes a success. Well, scaling up the front-end is trivial.

Simply change your configuration file and recreate the application:

```
{
  "app_name": "guesser",
  "services": [
    {
      "service_name": "guesser-game",
      "components": [
        {
          "component_name": "guesser-front-end",
          "image": "arangodb/example-guesser",
          "ports": [ 8000 ],
          "dependencies": [
            { "name": "guesser-back-end", "port": 8529 }
          ],
          "domains": { "guesser.gigantic.io": 8000 },
          "scaling_policy": { "min": 2, "max": 2 }
        },
        {
          "component_name": "guesser-back-end",
          "image": "arangodb/example-guesser-db",
          "ports": [ 8529 ]
        }
      ]
    }
  ]
}
```

The important line is

```
"scaling_policy": { "min": 2, "max": 2 }
```

It tells the swarm to use two front-end containers. In later version of the `swarm` you will be able to change the number of containers in a running application with the command:

```
> swarm scaleup guesser/guesser-game/guesser-front-end --count=1
Scaling up component guesser/guesser-game/guesser-front-end by 1...
```

We at ArangoDB are hard at work to make scaling up the back-end database equally easy. Stay tuned for new releases in early 2015...

Authors: Frank Celler

Tags: #docker, #giantswarm, #howto

ArangoDB on Apache Mesos using Marathon and Docker

Problem

I want to use ArangoDB in Apache Mesos with Docker containers.

Solution

Mesos in its newest version makes it very easy to use ArangoDB, because Mesos has added support for docker containers. Together with Marathon to start the front-end and back-end parts of an application, installation is straight forward.

My colleague Max has written a guesser game with various front-ends and ArangoDB as backend. In order to get the feeling of being part of the Mesosphere, I have started to set up this game in an DigitalOcean environment.

First Steps

The guesser game consists of a front-end written as express application in node and a storage back-end using ArangoDB and a small API developed with the Foxx microservices framework.

The front-end application is available as image

```
arangodb/example-guesser
```

and the ArangoDB back-end with the Foxx API as

```
arangodb/example-guesser-db
```

The dockerfiles used to create the images are available from github

```
https://github.com/arangodb/guesser
```

Set Up the Environment

Follow the instructions on Mesosphere to setup an environment with docker support. You should end up with ssh access to the Mesos master.

Set Up the Application

For this tutorial we bind the database to a fixed port on the Mesos environment. Please note, that the mesosphere uses HAproxy to map the global port to the real host and port. The servers created by Mesosphere will have a HAproxy defined on all masters and slaves.

That means, if we chose `32333` as service port for the database, it will be reachable on this port on all masters and slaves. The app definition for the database looks like

```
{
  "id": "/guesser/database",
  "apps": [
     {
       "id": "/guesser/database/arangodb",
       "container": {
         "docker": {
           "image": "arangodb/example-guesser-db",
           "network": "BRIDGE",
           "portMappings": [
               { "containerPort": 8529, "hostPort": 0, "servicePort": 32222, "protocol": "tcp" }
           ]
         }
       },
```

```
      "cpus": 0.2,
      "mem": 512.0,
      "instances": 1
    }
  ]
}
```

This will start the docker image for the back-end and binds the port to 32222.

Inside the docker container an environment variable `HOST` is set be the mesos slave to point to the slave. The front-end can therefore access port `32222` on this host to contact the HAproxy, gaining access to the database.

The app definition for the front-end looks like

```
{
  "id": "/guesser/frontend",
  "apps": [
    {
      "id": "/guesser/frontend/node",
      "container": {
        "docker": {
          "image": "arangodb/example-guesser",
          "network": "BRIDGE",
          "portMappings": [
              { "containerPort": 8000, "hostPort": 0, "servicePort": 32221, "protocol": "tcp" }
          ]
        }
      },
      "cpus": 0.2,
      "mem": 256.0,
      "instances": 1
    }
  ]
}
```

Marathon allows to define a group of applications with dependencies between the components. The front-end depends on the back-end, therefore the complete group definitions looks like

```
{
  "id": "/guesser",
  "groups": [
  {
    "id": "/guesser/database",
    "apps": [
     {
      "id": "/guesser/database/arangodb",
      "container": {
      "docker": {
        "image": "arangodb/example-guesser-db",
        "network": "BRIDGE",
        "portMappings": [
          { "containerPort": 8529, "hostPort": 0, "servicePort": 32222, "protocol": "tcp" }
        ]
      }
      },
      "cpus": 0.2,
      "mem": 512.0,
      "instances": 1
    }
    ]
  },
  {
    "id": "/guesser/frontend",
    "dependencies": ["/guesser/database"],
    "apps": [
     {
      "id": "/guesser/frontend/node",
      "container": {
      "docker": {
        "image": "arangodb/example-guesser",
        "network": "BRIDGE",
        "portMappings": [
```

```
            { "containerPort": 8000, "hostPort": 0, "servicePort": 32221, "protocol": "tcp" }
          ]
        }
        },
        "cpus": 0.2,
        "mem": 256.0,
        "instances": 1
      }
     ]
   }
   ]
}
```

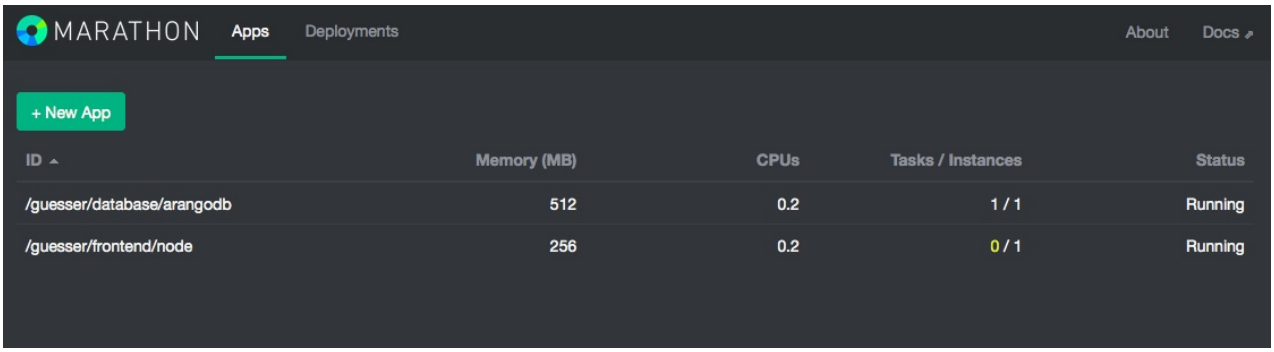
This starts one instance of the back-end called `/guesser/database/arangodb` and one instance of the front-end called `/guesser/frontend/node`. The front-end depends on the back-end.

In order to fire up the guesser game save the above definition in a file `guesser.json` and execute

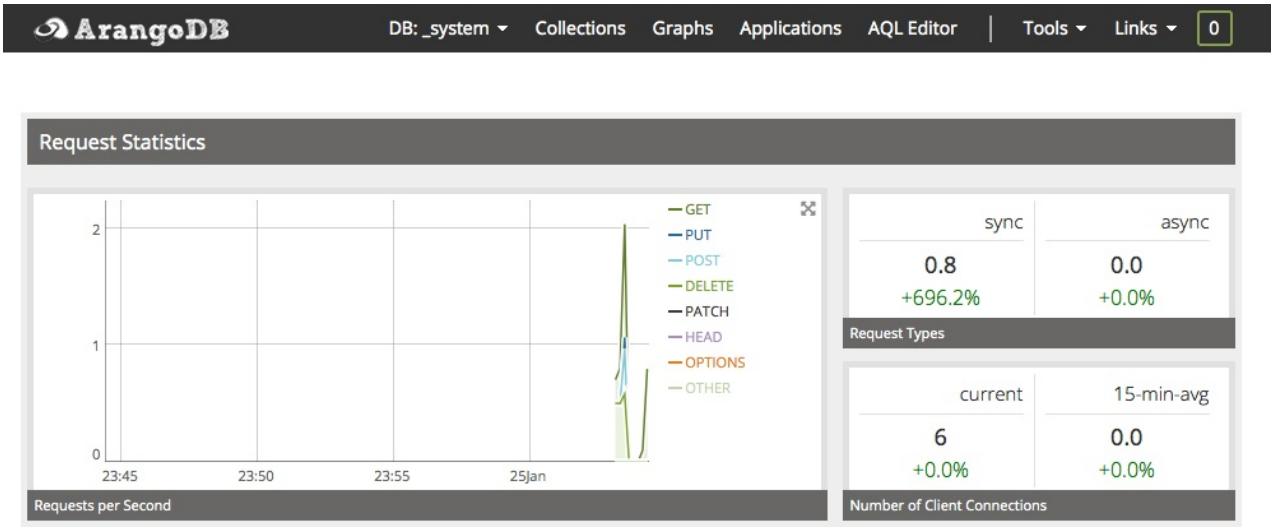
```
curl -X PUT -H "Accept: application/json" -H "Content-Type: application/json"  127.0.0.1:8080/v2/groups -d "`cat guesser.json`"
```

on the mesos master.

If you now switch to the Marathon console on port `8080`, you should see apps, namely `/guesser/database/arangodb` and `/guesser/frontend/node`.



If you access port `32222`, you should see the ArangoDB console.



And finally, on port `32211`, you can play the guesser game.

Scaling Up

Your game becomes a success. Well, scaling up the front-end is trivial. Simply, go to the marathon page and scale up `/guesser/frontend/node`.

Authors: Frank Celler

Tags: #docker, #mesos, #mesosphere, #howto

Deploying a highly available application using ArangoDB and Foxx on DC/OS

Problem

How can I deploy an application using ArangoDB on DC/OS and make everything highly available?

Solution

To achieve this goal several individual components have to be combined.

Install DC/OS

Go to https://dcos.io/install/ and follow the instructions to install DC/OS

Also make sure to install the dcos CLI.

Install ArangoDB

Once your cluster is DC/OS cluster is ready install the package `arangodb3` from the universe tab (default settings are fine)

Detailed instructions may be found in the first chapters of our DC/OS tutorial here:

https://dcos.io/docs/1.7/usage/tutorials/arangodb/

To understand how ArangoDB ensures that it is highly available make sure to read the cluster documentation here:

ArangoDB Architecture Documentation

Deploy a load balancer for the coordinators

Once ArangoDB is installed and healthy you can access the cluster via one of the coordinators.

To do so from the outside DC/OS provides a nice and secure gateway through their admin interface.

However this is intended to be used from the outside only. Applications using ArangoDB as its data store will want to connect to the coordinators from the inside. You could check the task list within DC/OS to find out the endpoints where the coordinators are listening. However these are not to be trusted: They can fail at any time, the number of coordinators might change due to up- and downscaling or someone might even kill a full DC/OS Agent and tasks may simply fail and reappear on a different endpoint.

In short: Endpoints are `temporary`.

To mitigate this problem we have built a special load balancer for these coordinators.

To install it:

```
$ git clone https://github.com/arangodb/arangodb-mesos-haproxy
$ cd arangodb-mesos-haproxy
$ dcos marathon app add marathon.json
```

Afterwards you can use the following endpoint to access the coordinators from within the cluster:

```
tcp://arangodb-proxy.marathon.mesos:8529
```

To make it highly available you can simply launch a few more instances using the marathon web interface. Details on how to do this and how to deploy an application using the UI can be found here: https://dcos.io/docs/1.7/usage/tutorials/marathon/marathon101/

Our test application

Now that we have setup ArangoDB on DC/OS it is time to deploy our application. In this example we will use our guesser application which you can find here:

https://github.com/arangodb/guesser

This application has some application code and a Foxx microservice.

Deploying the Foxx service

Open the ArangoDB interface (via the `Services` tab in the DC/OS interface) and go to `Services`.

Enter `/guesser` as mount directory Choose `github` on the tab and enter the following repository:

```
ArangoDB/guesser
```

Choose `master` as version.

Press `Install`

Deploy the application

Finally it is time to deploy the application code. We have packaged everything into a docker container. The only thing that is missing is some connection info for the database. This can be provided via environment variables through marathon.

Open the marathon webinterface in your DC/OS cluster (`Services` and then `marathon`).

Then click `Create application`

On the top right you can change to the JSON view. Paste the following config:

```
{
  "id": "/guesser",
  "cmd": null,
  "cpus": 1,
  "mem": 128,
  "disk": 0,
  "instances": 3,
  "container": {
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "arangodb/example-guesser",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 8000,
          "hostPort": 0,
          "servicePort": 10004,
          "protocol": "tcp"
        }
      ],
      "privileged": false,
      "parameters": [],
      "forcePullImage": true
    }
  },
  "labels":{
    "HAPROXY_GROUP":"external"
  },
  "env": {
    "ARANGODB_SERVER": "http://arangodb-proxy.marathon.mesos:8529",
    "ARANGODB_ENDPOINT": "tcp://arangodb-proxy.marathon.mesos:8529"
  }
}
```

As you can see we are providing the `ARANGODB_ENDPOINT` as an environment variable. The docker container will take that and use it when connecting. This configuration injection via environment variables is considered a docker best practice and this is how you should probably create your applications as well.

Now we have our guesser app started within mesos.

It is highly available right away as we launched 3 instances. To scale it up or down simply use the scale buttons in the marathon UI.

Make it publically available

For this to work we need another tool, namely `marathon-lb`

Install it:

```
dcos package install marathon-lb
```

After installation it will scan all marathon applications for a special set of labels and make these applications available to the public.

To make the guesser application available to the public you first have to determine a hostname that points to the external loadbalancer in your environment. When installing using the cloudformation template on AWS this is the hostname of the so called `public slave`. You can find it in the output tab of the cloudformation template.

In my case this was:

`mop-publicslaveloa-3phq11mb7oez-1979417947.eu-west-1.elb.amazonaws.com`

In case there are uppercased letters please take *extra care* to lowercase them as the marathon-lb will fail otherwise.

Edit the settings of the guesser app in the marathon UI and add this hostname as the label HAPROXY_0_VHOST either using the UI or using the JSON mode:

```
[...]
  "labels": {
    "HAPROXY_GROUP": "external",
    "HAPROXY_0_VHOST": "mop-publicslaveloa-3phq11mb7oez-1979417947.eu-west-1.elb.amazonaws.com"
  },
[...]
```

To scale it up and thus making it highly available increase the `instances` count within marathon.

For more detailed information and more configuration options including SSL etc be sure to check the documentation:

https://docs.mesosphere.com/1.7/usage/service-discovery/marathon-lb/usage/

Accessing the guesser game

After marathon-lb has reloaded its configuration (which should happen almost immediately) you should be able to access the guesser game by pointing a web browser to your hostname.

Have fun!

Conclusion

There are quite a few components involved in this process but once you are finished you will have a highly resilient system which surely was worth the effort. None of the components involved is a single point of failure.

Author: Andreas Streichardt

Tags: #docker #howto #dcos #cluster #ha

Monitoring ArangoDB using collectd

Problem

The ArangoDB web interface shows a nice summary of the current state. I want to see similar numbers in my monitoring system so I can analyze the system usage post mortem or send alarms on failure.

Solution

Collectd is an excellent tool to gather all kinds of metrics from a system and deliver it to a central monitoring like Graphite and / or Nagios.

Ingredients

For this recipe you need to install the following tools:

- collectd >= 5.4.2 The aggregation Daemon
- kcollectd for inspecting the data

Configuring collectd

For aggregating the values we will use the cURL-JSON plug-in. We will store the values using the Round-Robin-Database writer(RRD) which `kcollectd` can later on present to you.

We assume your `collectd` comes from your distribution and reads its config from `/etc/collectd/collectd.conf`. Since this file tends to become pretty unreadable quickly, we use the `include` mechanism:

```
<Include "/etc/collectd/collectd.conf.d">
  Filter "*.conf"
</Include>
```

This way we can make each metric group on compact set config files. It consists of three components:

- loading the plug-in
- adding metrics to the TypesDB
- the configuration for the plug-in itself

rrdtool

We will use the Round-Robin-Database as storage backend for now. It creates its own database files of fixed size for each specific time range. Later you may choose more advanced writer-plug-ins, which may do network distribution of your metrics or integrate the above mentioned Graphite or your already established monitoring, etc.

For the RRD we will go pretty much with defaults:

```
# Load the plug-in:
LoadPlugin rrdtool
<Plugin rrdtool>
   DataDir "/var/lib/collectd/rrd"
#  CacheTimeout 120
#  CacheFlush 900
#  WritesPerSecond 30
#  CreateFilesAsync false
#  RandomTimeout 0
#
# The following settings are rather advanced
# and should usually not be touched:
#   StepSize 10
#   HeartBeat 20
```

```
#   RRARows 1200
#   RRATimespan 158112000
#   XFF 0.1
</Plugin>
```

cURL JSON

`Collectd` comes with a wide range of metric aggregation plug-ins. Many tools today use JSON as data formating grammar; so does ArangoDB. Therefore a plug-in offering to fetch JSON documents via HTTP is the perfect match as an integration interface:

```
# Load the plug-in:
LoadPlugin curl_json
# we need to use our own types to generate individual names for our gauges:
TypesDB "/etc/collectd/collectd.conf.d/arangodb_types.db"
<Plugin curl_json>
  # Adjust the URL so collectd can reach your arangod:
  <URL "http://localhost:8529/_db/_system/_admin/aardvark/statistics/short">
   # Set your authentication to Aardvark here:
   # User "foo"
   # Password "bar"
    <Key "totalTimeDistributionPercent/values/0">
       Type "totalTimeDistributionPercent_values"
     </Key>
     <Key "totalTimeDistributionPercent/cuts/0">
       Type "totalTimeDistributionPercent_cuts"
     </Key>
     <Key "requestTimeDistributionPercent/values/0">
       Type "requestTimeDistributionPercent_values"
     </Key>
     <Key "requestTimeDistributionPercent/cuts/0">
       Type "requestTimeDistributionPercent_cuts"
     </Key>
     <Key "queueTimeDistributionPercent/values/0">
       Type "queueTimeDistributionPercent_values"
     </Key>
     <Key "queueTimeDistributionPercent/cuts/0">
       Type "queueTimeDistributionPercent_cuts"
     </Key>
     <Key "bytesSentDistributionPercent/values/0">
       Type "bytesSentDistributionPercent_values"
     </Key>
     <Key "bytesSentDistributionPercent/cuts/0">
       Type "bytesSentDistributionPercent_cuts"
     </Key>
     <Key "bytesReceivedDistributionPercent/values/0">
       Type "bytesReceivedDistributionPercent_values"
     </Key>
     <Key "bytesReceivedDistributionPercent/cuts/0">
       Type "bytesReceivedDistributionPercent_cuts"
     </Key>
     <Key "numberOfThreadsCurrent">
       Type "gauge"
     </Key>
     <Key "numberOfThreadsPercentChange">
       Type "gauge"
     </Key>
     <Key "virtualSizeCurrent">
       Type "gauge"
     </Key>
     <Key "virtualSizePercentChange">
       Type "gauge"
     </Key>
     <Key "residentSizeCurrent">
       Type "gauge"
     </Key>
     <Key "residentSizePercent">
       Type "gauge"
     </Key>
     <Key "asyncPerSecondCurrent">
       Type "gauge"
     </Key>
     <Key "asyncPerSecondPercentChange">
       Type "gauge"
```

```
      </Key>
      <Key "syncPerSecondCurrent">
        Type "gauge"
      </Key>
      <Key "syncPerSecondPercentChange">
        Type "gauge"
      </Key>
      <Key "clientConnectionsCurrent">
        Type "gauge"
      </Key>
      <Key "clientConnectionsPercentChange">
        Type "gauge"
      </Key>
      <Key "physicalMemory">
        Type "gauge"
      </Key>
      <Key "nextStart">
        Type "gauge"
      </Key>
      <Key "waitFor">
        Type "gauge"
      </Key>
      <Key "numberOfThreads15M">
        Type "gauge"
      </Key>
      <Key "numberOfThreads15MPercentChange">
        Type "gauge"
      </Key>
      <Key "virtualSize15M">
        Type "gauge"
      </Key>
      <Key "virtualSize15MPercentChange">
        Type "gauge"
      </Key>
      <Key "asyncPerSecond15M">
        Type "gauge"
      </Key>
      <Key "asyncPerSecond15MPercentChange">
        Type "gauge"
      </Key>
      <Key "syncPerSecond15M">
        Type "gauge"
      </Key>
      <Key "syncPerSecond15MPercentChange">
        Type "gauge"
      </Key>
      <Key "clientConnections15M">
        Type "gauge"
      </Key>
      <Key "clientConnections15MPercentChange">
        Type "gauge"
      </Key>
    </URL>
</Plugin>
```

To circumvent the shortcoming of the curl_JSON plug-in to only take the last path element as name for the metric, we need to give them a name using our own `types.db` file in `/etc/collectd/collectd.conf.d/arangodb_types.db`:

```
totalTimeDistributionPercent_values        value:GAUGE:U:U
totalTimeDistributionPercent_cuts        value:GAUGE:U:U
requestTimeDistributionPercent_values        value:GAUGE:U:U
requestTimeDistributionPercent_cuts        value:GAUGE:U:U
queueTimeDistributionPercent_values        value:GAUGE:U:U
queueTimeDistributionPercent_cuts        value:GAUGE:U:U
bytesSentDistributionPercent_values        value:GAUGE:U:U
bytesSentDistributionPercent_cuts        value:GAUGE:U:U
bytesReceivedDistributionPercent_values        value:GAUGE:U:U
bytesReceivedDistributionPercent_cuts        value:GAUGE:U:U
```

Rolling your own

You may want to monitor your own metrics from ArangoDB. Here is a simple example how to use the `config`:

```
{
 "testArray":[1,2],
 "testArrayInbetween":[{"blarg":3},{"blub":4}],
 "testDirectHit":5,
 "testSubLevelHit":{"oneMoreLevel":6}
}
```

This `config` snippet will parse the JSON above:

```
<Key "testArray/0">
  Type "gauge"
  # Expect: 1
</Key>
<Key "testArray/1">
  Type "gauge"
  # Expect: 2
</Key>
<Key "testArrayInbetween/0/blarg">
  Type "gauge"
  # Expect: 3
</Key>
<Key "testArrayInbetween/1/blub">
  Type "gauge"
  # Expect: 4
</Key>
<Key "testDirectHit">
  Type "gauge"
  # Expect: 5
</Key>
<Key "testSubLevelHit/oneMoreLevel">
  Type "gauge"
  # Expect: 6
</Key
```

Get it served

Now we will (re)start `collectd` so it picks up our configuration:

```
/etc/init.d/collectd start
```

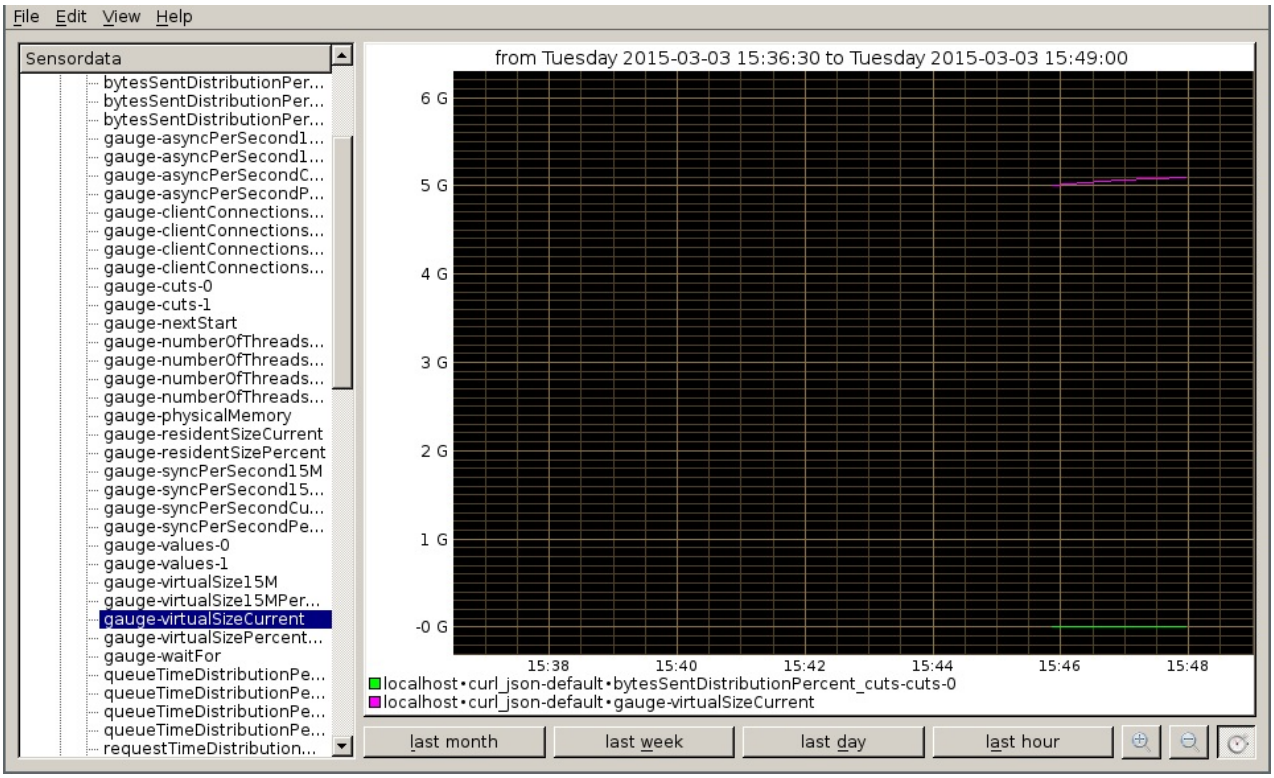
We will inspect the syslog to revalidate nothing went wrong:

```
Mar  3 13:59:52 localhost collectd[11276]: Starting statistics collection and monitoring daemon: collectd.
Mar  3 13:59:52 localhost systemd[1]: Started LSB: manage the statistics collection daemon.
Mar  3 13:59:52 localhost collectd[11283]: Initialization complete, entering read-loop.
```

`Collectd` adds the hostname to the directory address, so now we should have files like these:

```
 -rw-r--r-- 1 root root 154888 Mar  2 16:53 /var/lib/collectd/rrd/localhost/curl_json-default/gauge-numberOfThreads15M.rrd
```

Now we start `kcollectd` to view the values in the RRD file:



Since we started putting values in just now, we need to choose 'last hour' and zoom in a little more to inspect the values.

Finished with this dish, wait for more metrics to come in other recipes.

Author: Wilfried Goesgens

Tags: #json #monitoring

Monitoring replication slave

Note: this recipe is working with ArangoDB 2.5, you need a collectd curl_json plugin with correct boolean type mapping.

Problem

How to monitor the slave status using the `collectd curl_JSON` plugin.

Solution

Since arangodb reports the replication status in JSON, integrating it with the collectd curl_JSON plugin should be an easy exercise. However, only very recent versions of collectd will handle boolean flags correctly.

Our test master/slave setup runs with the the master listening on `tcp://127.0.0.1:8529` and the slave (which we query) listening on `tcp://127.0.0.1:8530`. They replicate a dabatase by the name `testDatabase`.

Since replication appliers are active per database and our example doesn't use the default `_system`, we need to specify its name in the URL like this: `_db/testDatabase`.

We need to parse a document from a request like this:

```
curl --dump - http://localhost:8530/_db/testDatabase/_api/replication/applier-state
```

If the replication is not running the document will look like that:

```
{
  "state": {
    "running": false,
    "lastAppliedContinuousTick": null,
    "lastProcessedContinuousTick": null,
    "lastAvailableContinuousTick": null,
    "safeResumeTick": null,
    "progress": {
      "time": "2015-11-02T13:24:07Z",
      "message": "applier shut down",
      "failedConnects": 0
    },
    "totalRequests": 1,
    "totalFailedConnects": 0,
    "totalEvents": 0,
    "totalOperationsExcluded": 0,
    "lastError": {
      "time": "2015-11-02T13:24:07Z",
      "errorMessage": "no start tick",
      "errorNum": 1413
    },
    "time": "2015-11-02T13:31:53Z"
  },
  "server": {
    "version": "2.7.0",
    "serverId": "175584498800385"
  },
  "endpoint": "tcp://127.0.0.1:8529",
  "database": "testDatabase"
}
```

A running replication will return something like this:

```
{
  "state": {
    "running": true,
    "lastAppliedContinuousTick": "1150610894145",
    "lastProcessedContinuousTick": "1150610894145",
```

```
    "lastAvailableContinuousTick": "1151639153985",
    "safeResumeTick": "1150610894145",
    "progress": {
      "time": "2015-11-02T13:49:56Z",
      "message": "fetching master log from tick 1150610894145",
      "failedConnects": 0
    },
    "totalRequests": 12,
    "totalFailedConnects": 0,
    "totalEvents": 2,
    "totalOperationsExcluded": 0,
    "lastError": {
      "errorNum": 0
    },
    "time": "2015-11-02T13:49:57Z"
  },
  "server": {
    "version": "2.7.0",
    "serverId": "175584498800385"
  },
  "endpoint": "tcp://127.0.0.1:8529",
  "database": "testDatabase"
}
```

We create a simple collectd configuration in `/etc/collectd/collectd.conf.d/slave_testDatabase.conf` that matches our API:

```
TypesDB "/etc/collectd/collectd.conf.d/slavestate_types.db"
<Plugin curl_json>
  # Adjust the URL so collectd can reach your arangod slave instance:
  <URL "http://localhost:8530/_db/testDatabase/_api/replication/applier-state">
   # Set your authentication to that database here:
   # User "foo"
   # Password "bar"
    <Key "state/running">
       Type "boolean"
     </Key>
    <Key "state/totalOperationsExcluded">
       Type "counter"
     </Key>
    <Key "state/totalRequests">
       Type "counter"
     </Key>
    <Key "state/totalFailedConnects">
       Type "counter"
     </Key>
  </URL>
</Plugin>
```

To get nice metric names, we specify our own `types.db` file in `/etc/collectd/collectd.conf.d/slavestate_types.db`:

```
boolean                 value:ABSOLUTE:0:1
```

So, basically `state/running` will give you `0` / `1` if its (not /) running through the collectd monitor.

Author: Wilfried Goesgens

Tags: #monitoring #foxx #json

Monitoring ArangoDB Cluster network usage

Problem

We run a cluster and want to know whether the traffic is unbalanced or something like that. We want a cheap estimate which host has how much traffic.

Solution

As we already run Collectd as our metric-hub, we want to utilize it to also give us these figures. A very cheap way to generate these values are the counters in the IPTables firewall of our system.

Ingredients

For this recipe you need to install the following tools:

- collectd: the aggregation Daemon
- kcollectd for inspecting the data
- iptables - should come with your Linux distribution
- ferm for compact firewall code
- we base on Monitoring with Collecd recipe for understanding the basics about collectd

Getting the state and the Ports of your cluster

Now we need to find out the current configuration of our cluster. For the time being we assume you simply issued

```
./scripts/startLocalCluster.sh
```

to get you set up. So you know you've got two DB-Servers - one Coordinator, one agent:

```
ps -eaf |grep arango
arangod    21406     1  1 16:59 pts/14   00:00:00 bin/etcd-arango --data-dir /var/tmp/tmp-21550-1347489353/shell_server/agentar
ango4001 --name agentarango4001 --bind-addr 127.0.0.1:4001 --addr 127.0.0.1:4001 --peer-bind-addr 127.0.0.1:7001 --peer-addr 12
7.0.0.1:7001 --initial-cluster-state new --initial-cluster agentarango4001=http://127.0.0.1:7001
arangod    21408     1  4 16:56 pts/14   00:00:01 bin/arangod --database.directory cluster/data8629 --cluster.agency-endpoint t
cp://localhost:4001 --cluster.my-address tcp://localhost:8629 --server.endpoint tcp://localhost:8629 --cluster.my-local-info db
server:localhost:8629 --log.file cluster/8629.log --cluster.my-id Pavel
arangod    21410     1  5 16:56 pts/14   00:00:02 bin/arangod --database.directory cluster/data8630 --cluster.agency-endpoint t
cp://localhost:4001 --cluster.my-address tcp://localhost:8630 --server.endpoint tcp://localhost:8630 --cluster.my-local-info db
server:localhost:8630 --log.file cluster/8630.log --cluster.my-id Perry
arangod    21416     1  5 16:56 pts/14   00:00:02 bin/arangod --database.directory cluster/data8530 --cluster.agency-endpoint t
cp://localhost:4001 --cluster.my-address tcp://localhost:8530 --server.endpoint tcp://localhost:8530 --cluster.my-local-info co
ordinator:localhost:8530 --log.file cluster/8530.log --cluster.my-id Claus
```

We can now check which ports they occupied:

```
netstat -aplnt |grep arango
tcp        0      0 127.0.0.1:7001          0.0.0.0:*               LISTEN      21406/etcd-arango
tcp        0      0 127.0.0.1:4001          0.0.0.0:*               LISTEN      21406/etcd-arango
tcp        0      0 127.0.0.1:8530          0.0.0.0:*               LISTEN      21416/arangod
tcp        0      0 127.0.0.1:8629          0.0.0.0:*               LISTEN      21408/arangod
tcp        0      0 127.0.0.1:8630          0.0.0.0:*               LISTEN      21410/arangod
```

- The agent has 7001 and 4001. Since it's running in single server mode its cluster port (7001) should not show any traffic, port 4001 is the interesting one.
- Claus - This is the coordinator. Your Application will talk to it on port 8530
- Pavel - This is the first DB-Server; Claus will talk to it on port 8629
- Perry - This is the second DB-Server; Claus will talk to it on port 8630

Configuring IPTables / ferm

Since the usual solution using shell scripts calling iptables brings the DRY principle to a grinding hold, we need something better. Here ferm comes to the rescue - It enables you to produce very compact and well readable firewall configurations.

According to the ports we found in the last section, we will configure our firewall in `/etc/ferm/ferm.conf`, and put the identities into the comments so we have a persistent naming scheme:

```
# blindly forward these to the accounting chain:
@def $ARANGO_RANGE=4000:9000;

@def &TCP_ACCOUNTING($PORT, $COMMENT, $SRCCHAIN) = {
    @def $FULLCOMMENT=@cat($COMMENT, "_", $SRCCHAIN);
    dport $PORT mod comment comment $FULLCOMMENT NOP;
}

@def &ARANGO_ACCOUNTING($CHAINNAME) = {
# The coordinators:
    &TCP_ACCOUNTING(8530, "Claus", $CHAINNAME);
# The db-servers:
    &TCP_ACCOUNTING(8629, "Pavel", $CHAINNAME);
    &TCP_ACCOUNTING(8630, "Perry", $CHAINNAME);
# The agency:
    &TCP_ACCOUNTING(4001, "etcd_client", $CHAINNAME);
# it shouldn't talk to itself if it is only running with a single instance:
    &TCP_ACCOUNTING(7007, "etcd_cluster", $CHAINNAME);
}

table filter {
    chain INPUT {
        proto tcp dport $ARANGO_RANGE @subchain "Accounting" {
            &ARANGO_ACCOUNTING("input");
        }
        policy DROP;

        # connection tracking
        mod state state INVALID DROP;
        mod state state (ESTABLISHED RELATED) ACCEPT;

        # allow local packet
        interface lo ACCEPT;

        # respond to ping
        proto icmp ACCEPT;

        # allow IPsec
        proto udp dport 500 ACCEPT;
        proto (esp ah) ACCEPT;

        # allow SSH connections
        proto tcp dport ssh ACCEPT;
    }
    chain OUTPUT {
        policy ACCEPT;

        proto tcp dport $ARANGO_RANGE @subchain "Accounting" {
            &ARANGO_ACCOUNTING("output");
        }

        # connection tracking
        #mod state state INVALID DROP;
        mod state state (ESTABLISHED RELATED) ACCEPT;
    }
    chain FORWARD {
        policy DROP;

        # connection tracking
        mod state state INVALID DROP;
        mod state state (ESTABLISHED RELATED) ACCEPT;
    }
}
```

Note: This is a very basic configuration, mainly with the purpose to demonstrate the accounting feature - so don't run this in production)

After activating it interactively with

```
ferm -i /etc/ferm/ferm.conf
```

We now use the iptables command line utility directly to review the status our current setting:

```
iptables -L -nvx
Chain INPUT (policy DROP 85 packets, 6046 bytes)
    pkts      bytes target     prot opt in     out     source               destination
    7636  1821798 Accounting  tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            tcp dpts:4000:9000
       0        0 DROP       all  --  *      *       0.0.0.0/0            0.0.0.0/0           state INVALID
   14700 14857709 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
     130     7800 ACCEPT     all  --  lo     *       0.0.0.0/0            0.0.0.0/0
       0        0 ACCEPT     icmp --  *      *       0.0.0.0/0            0.0.0.0/0
       0        0 ACCEPT     udp  --  *      *       0.0.0.0/0            0.0.0.0/0           udp dpt:500
       0        0 ACCEPT     esp  --  *      *       0.0.0.0/0            0.0.0.0/0
       0        0 ACCEPT     ah   --  *      *       0.0.0.0/0            0.0.0.0/0
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:22

Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 DROP       all  --  *      *       0.0.0.0/0            0.0.0.0/0           state INVALID
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED

Chain OUTPUT (policy ACCEPT 296 packets, 19404 bytes)
    pkts      bytes target     prot opt in     out     source               destination
    7720  1882404 Accounting  tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            tcp dpts:4000:9000
   14575 14884356 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED

Chain Accounting (2 references)
    pkts      bytes target     prot opt in     out     source               destination
     204    57750            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8530 /* Claus_input */
      20    17890            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8629 /* Pavel_input */
     262    97352            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8630 /* Perry_input */
    2604   336184            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:4001 /* etcd_client_inpu
t */
       0        0            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:7007 /* etcd_cluster_inp
ut */
     204    57750            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8530 /* Claus_output */
      20    17890            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8629 /* Pavel_output */
     262    97352            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8630 /* Perry_output */
    2604   336184            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:4001 /* etcd_client_outp
ut */
       0        0            tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:7007 /* etcd_cluster_out
put */
```

You can see nicely the Accounting sub-chain with our comments. These should be pretty straight forward to match. We also see the **pkts** and **bytes** columns. They contain the current value of these counters of your system.

Read more about linux firewalling and ferm configuration to be sure you do the right thing.

Configuring Collectd to pick up these values

Since your system now generates these numbers, we want to configure collectd with its iptables plugin to aggregate them.

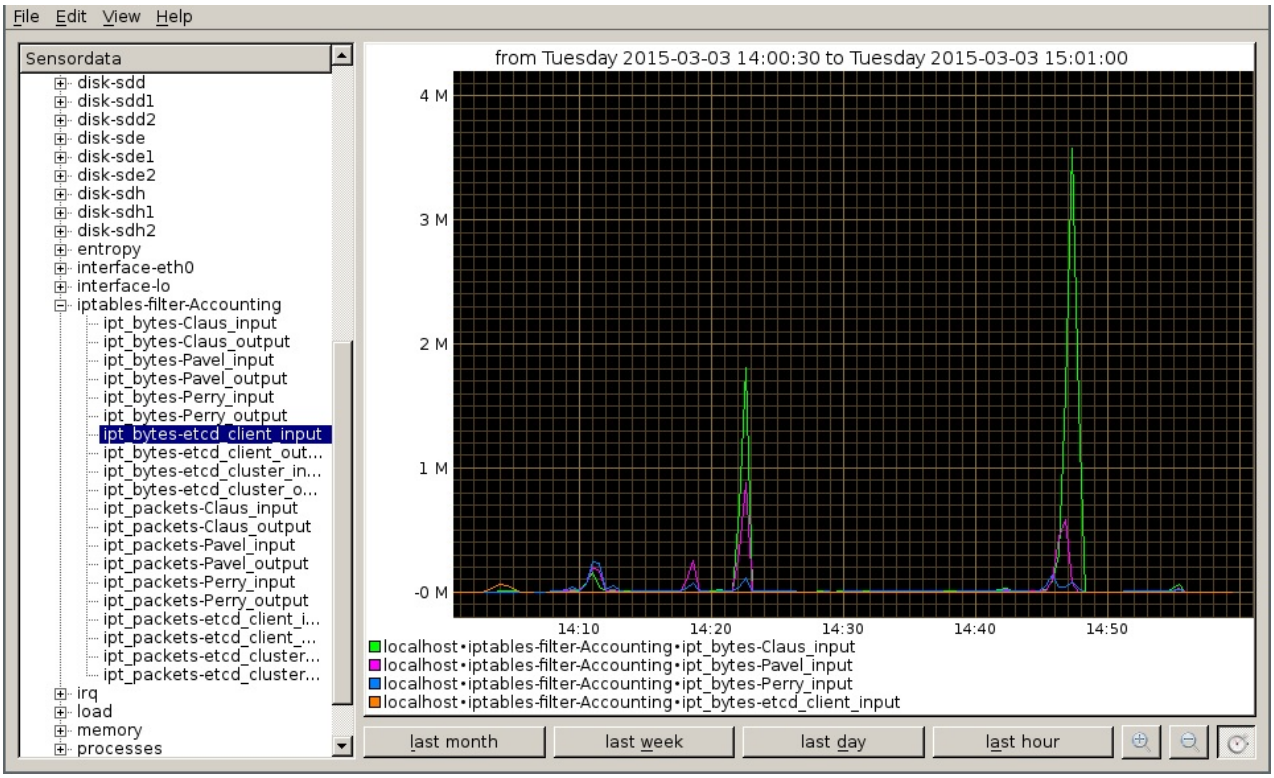
We do so in the `/etc/collectd/collectd.conf.d/iptables.conf` :

```
LoadPlugin iptables
<Plugin iptables>
  Chain filter "Accounting" "Claus_input"
  Chain filter "Accounting" "Pavel_input"
  Chain filter "Accounting" "Perry_input"
  Chain filter "Accounting" "etcd_client_input"
  Chain filter "Accounting" "etcd_cluster_input"
  Chain filter "Accounting" "Claus_output"
  Chain filter "Accounting" "Pavel_output"
  Chain filter "Accounting" "Perry_output"
  Chain filter "Accounting" "etcd_client_output"
  Chain filter "Accounting" "etcd_cluster_output"
</Plugin>
```

Now we restart collectd with `/etc/init.d/collectd restart`, watch the syslog for errors. If everything is OK, our values should show up in:

```
/var/lib/collectd/rrd/localhost/iptables-filter-Accounting/ipt_packets-Claus_output.rrd
```

We can inspect our values with kcollectd:



Author: Wilfried Goesgens

Tags: #monitoring

Monitoring other relevant metrics of ArangoDB

Problem

Aside of the values which ArangoDB already offers for monitoring, other system metrics may be relevant for continuously operating ArangoDB. be it a single instance or a cluster setup. Collectd offers a pleathora of plugins - lets have a look at some of them which may be useful for us.

Solution

Ingedients

For this recipe you need to install the following tools:

- collectd: The metrics aggregation Daemon
- we base on Monitoring with Collecd recipe for understanding the basics about collectd

Disk usage

You may want to monitor that ArangoDB doesn't run out of disk space. The df Plugin can aggregate these values for you.

First we need to find out which disks are used by your ArangoDB. By default you need to find **/var/lib/arango** in the mountpoints. Since nowadays many virtual file systems are also mounted on a typical *nix system we want to sort the output of mount:

```
mount | sort
/dev/sda3 on /local/home type ext4 (rw,relatime,data=ordered)
/dev/sda4 on / type ext4 (rw,relatime,data=ordered)
/dev/sdb1 on /mnt type vfat (rw,relatime,fmask=0022,dmask=0022,codepage=437,iocharset=utf8,shortname=mixed,errors=remount-ro)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,relatime)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
....
udev on /dev type devtmpfs (rw,relatime,size=10240k,nr_inodes=1022123,mode=755)
```

So here we can see the mountpoints are `/`, `/local/home`, `/mnt/` so `/var/lib/` can be found on the root partition (`/`) `/dev/sda3` here. A production setup may be different so the OS doesn't interfere with the services.

The collectd configuration `/etc/collectd/collectd.conf.d/diskusage.conf` looks like this:

```
LoadPlugin df
<Plugin df>
  Device "/dev/sda3"
  #  Device "192.168.0.2:/mnt/nfs"
  #  MountPoint "/home"
  #  FSType "ext4"
  #  ignore rootfs; else, the root file-system would appear twice, causing
  #  one of the updates to fail and spam the log
  FSType rootfs
  # ignore the usual virtual / temporary file-systems
  FSType sysfs
  FSType proc
  FSType devtmpfs
  FSType devpts
  FSType tmpfs
  FSType fusectl
  FSType cgroup
  IgnoreSelected true
  #  ReportByDevice false
  #  ReportReserved false
  #  ReportInodes false
  #  ValuesAbsolute true
  #  ValuesPercentage false
</Plugin>
```

Disk I/O Usage

Another interesting metric is the amount of data read/written to disk - its an estimate how busy your ArangoDB or the whole system currently is. The Disk plugin aggregates these values.

According to the mountpoints above our configuration `/etc/collectd/collectd.conf.d/disk_io.conf` looks like this:

```
LoadPlugin disk
<Plugin disk>
  Disk "hda"
  Disk "/sda[23]/"
  IgnoreSelected false
</Plugin>
```

CPU Usage

While the ArangoDB self monitoring already offers some overview of the running threads etc. you can get a deeper view using the Process Plugin.

If you're running a single Arango instance, a simple match by process name is sufficient, `/etc/collectd/collectd.conf.d/arango_process.conf` looks like this:

```
LoadPlugin processes
<Plugin processes>
  Process "arangod"
</Plugin>
```

If you're running a cluster, you can match the specific instances by command-line parameters, `/etc/collectd/collectd.conf.d/arango_cluster.conf` looks like this:

```
LoadPlugin processes
<Plugin processes>
  ProcessMatch "Claus" "/usr/bin/arangod .*--cluster.my-id Claus.*"
  ProcessMatch "Pavel" "/usr/bin/arangod .*--cluster.my-id Pavel.*"
  ProcessMatch "Perry" "/usr/bin/arangod .*--cluster.my-id Perry.*"
  Process "etcd-arango"
</Plugin>
```

More Plugins

As mentioned above, the list of available plugins is huge; Here are some more one could be interested in:

- use the CPU Plugin to monitor the overall CPU utilization
- use the Memory Plugin to monitor main memory availability
- use the Swap Plugin to see whether excess RAM usage forces the system to page and thus slow down
- Ethernet Statistics with whats going on at your Network cards to get a more broad overview of network traffic
- you may Tail logfiles like an apache request log and pick specific requests by regular expressions
- Parse tabular files in the `/proc` file system
- you can use filters to reduce the amount of data created by plugins (i.e. if you have many CPU cores, you may want the combined result). It can also decide where to route data and to which writer plugin
- while you may have seen that metrics are stored at a fixed rate or frequency, your metrics (i.e. the durations of web requests) may come in a random & higher frequency. Thus you want to burn them down to a fixed frequency, and know Min/Max/Average/Median. So you want to Aggregate values using the statsd pattern.
- You may start rolling your own in Python, java, Perl or for sure in C, the language collectd is implemented in

Finally while kcollectd is nice to get a quick success at inspecting your collected metrics during working your way into collectd, its not as sufficient for operating a production site. Since collectds default storage RRD is already widespread in system monitoring, there are many webfrontents to choose for the visualization. Some of them replace the RRD storage by simply adding a writer plugin, most prominent the Graphite graphing framework with the Graphite writer which allows you to combine random metrics in single graphs - to find coincidences in your data you never dreamed of.

If you already run Nagios you can use the Nagios tool to submit values.

We hope you now have a good overview of whats possible, but as usual its a good idea to browse the Fine Manual.

Author: Wilfried Goesgens

Tags: #monitoring

Monitoring your Foxx applications

Note: this recipe is working with ArangoDB 2.5 Foxx

Problem

How to integrate a Foxx application into a monitoring system using the `collectd curl_JSON` plugin.

Solution

Since Foxx native tongue is JSON, integrating it with the collectd curl_JSON plugin should be an easy exercise. We have a Foxx-Application which can receive Data and write it into a collection. We specify an easy input Model:

```
Model = Foxx.Model.extend({
  schema: {
    // Describe the attributes with Joi here
    '_key': Joi.string(),
    'value': Joi.number()
  }
});
```

And use a simple Foxx-Route to inject data into our collection:

```
/** Creates a new FirstCollection
 *
 * Creates a new FirstCollection-Item. The information has to be in the
 * requestBody.
 */
controller.post('/firstCollection', function (req, res) {
  var firstCollection = req.params('firstCollection');
  firstCollection.attributes.Date = Date.now();
  res.json(FirstCollection_repo.save(firstCollection).forClient());
}).bodyParam('firstCollection', {
  description: 'The FirstCollection you want to create',
  type: FirstCollection
});
```

Which we may do using `cURL` :

```
echo '{"value":1 ,"_key":"13"}' | \
  curl -d @-  http://localhost:8529/_db/_system/collectable_foxx/data/firstCollection/firstCollection
```

We'd expect the value to be in the range of 1 to 5. Maybe the source of this data is a web-poll or something similar.

We now add another Foxx-route which we want to link with `collectd` :

```
/**
 * we use a group-by construct to get the values:
 */
var db = require('org/arangodb').db;
var searchQuery = 'FOR x IN @@collection FILTER x.Date >= @until collect value=x.value with count into counter RETURN {[[CONCAT
("choice", value)] : counter }';
controller.get('/firstCollection/lastSeconds/:nSeconds', function (req, res) {
  var until = Date.now() - req.params('nSeconds') * 1000;
  res.json(
    db._query(searchQuery, {
      '@collection': FirstCollection_repo.collection.name(),
      'until': until
    }).toArray()
  );
}).pathParam('nSeconds', {
  description: 'look up to n Seconds into the past',
```

```
    type: joi.string().required()
});
```

We inspect the return document using curl and jq for nice formatting:

```
curl 'http://localhost:8529/_db/_system/collectable_foxx/data/firstCollection/firstCollection/lastSeconds/10' |jq "."

[
  {
    "1": 3
    "3": 7
  }
]
```

We have to design the return values in a way that collectd's config syntax can simply grab it. This Route returns an object with flat key values where keys may range from 0 to 5. We create a simple collectd configuration in `/etc/collectd/collectd.conf.d/foxx_simple.conf` that matches our API:

```
# Load the plug-in:
LoadPlugin curl_json
# we need to use our own types to generate individual names for our gauges:
TypesDB "/etc/collectd/collectd.conf.d/foxx_simple_types.db"
<Plugin curl_json>
  # Adjust the URL so collectd can reach your arangod:
  <URL "http://localhost:8529/_db/_system/collectable_foxx/data/firstCollection/firstCollection/lastSeconds/10">
   # Set your authentication to Aardvark here:
   # User "foo"
   # Password "bar"
    <Key "choice0">
       Type "the_values"
     </Key>
    <Key "choice1">
       Type "first_values"
     </Key>
    <Key "choice2">
       Type "second_values"
     </Key>
    <Key "choice3">
       Type "third_values"
     </Key>
    <Key "choice4">
       Type "fourth_values"
     </Key>
    <Key "choice5">
       Type "fifth_values"
     </Key>
  </URL>
</Plugin>
```

To get nice metric names, we specify our own `types.db` file in `/etc/collectd/collectd.conf.d/foxx_simple_types.db`:

```
the_values    value:GAUGE:U:U
first_values    value:GAUGE:U:U
second_values    value:GAUGE:U:U
third_values    value:GAUGE:U:U
fourth_values    value:GAUGE:U:U
fifth_values    value:GAUGE:U:U
```

Author: Wilfried Goesgens

Tags: #monitoring #foxx #json