(FP7 614100)

D5.1.1 Initial Data Analysis & Knowledge Repository Technical Specifications & Guidelines

31 October 2014 – Version 1.2

Published by the IMPReSS Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme and the Conselho Nacional de Desenvolvimento Científico e Tecnológico
Objective ICT-2013.10.2 EU-Brazil research and development Cooperation

Document control page

Document file: D5.1.1_Initial_Data_Analysis_&_Knowledge_Repository_Technical_Specifications_&_Guidelines.doc
Document version: 1.2
Document owner: Djamel Sadok (UFPE)

Work package: WP5 – Data Storage, Analysis & Decision Support
Task: T5.1 Data and knowledge management support
Deliverable type: R

Document status: [X] approved by the document owner for internal review
[X] approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Eduardo Souto (UFAM)	20/02/2014	First Draft.
0.2	Lucas Lira Gomes (UFPE)	27/02/2014	Revised the document.
0.3	Lucas Lira Gomes (UFPE)	28/02/2014	Revamped section 2.
0.4	Eduardo Souto (UFAM)	28/02/2014	Document sent for internal review.
1.0	Djamel Sadok (UFPE)	03/03/2014	Final editing. Final version submitted to the European Commission.
1.1	Lucas Lira Gomes (UFPE)	16/03/2014	Addressed the concerns of the reviewers. Fixed minor typos.
1.2	Walter Andrade (UFPE)	31/10/2014	Major changes in the gremlin DSL documentation.

Internal review history:

Reviewed by	Date	Summary of comments
Carlos Kamienski (UFABC)	05/03/2014	Accepted with minor corrections and comments
Jussi Kiljander (VTT)	05/03/2014	Accepted with minor corrections and comments

Table of Contents

Executive summary **4**

Introduction **5**

1. IMPReSS Platform Overview **6**

2. Data, Policy and Knowledge Storage Module **7**

2.1 Data Model 7
2.2 Proposed Architecture 9

3. Domain Specific Language – DSL **13**

3.1 Impress' DSL Documentation 13
3.1.1 Impress' DSL Steps 28
3.1.2 Gremlim's Useful Steps 30
3.1.3 Impress' DSL Functions 31

4. Initial Performance Evaluation **32**

References **33**

Executive summary

This deliverable describes the initial technical specification of the data and knowledge repository used by the IMPReSS cloud. In the IMPReSS cloud, data semantics and analytics are fundamental to the the decision making process. Effectively managing the storage resources and data in the cloud is of paramount importance to maintaining satisfactory service levels.

In that sense, this deliverable aims to provide the foundations for a distributed and scalable storage solution for the IMPReSS cloud.

Introduction

The Internet of Things (IoT) is a concept that encloses a plethora of technologies and their applications, providing the means to access and control all kinds of smart devices (also named as "things"). IoT covers a wide range of objects, such as sensors, actuators, mobile devices, industrial controllers, HVAC (heating, ventilation, air-conditioning) units, household appliances like smart TVs and refrigerators, and so on. Radio Frequency Identification (RFID) and sensor network technologies have often been used to measure, infer and understand environmental indicators around us. This often results in the generation of huge volumes of data that have to be stored, processed and presented in a seamless, efficient, and easily interpretable form [1]. To meet these requirements, the Cloud Computing technologies can be used as an infrastructure for the storage, processing and computing of such massive amounts of data generated by highly distributed smart devices (e.g. sensors and actuators).

It is in this context, that this deliverable presents the initial technical specification of the data and knowledge repository adopted for use within the IMPReSS project [2]. The main idea is to provide a cloud infrastructure able to manage very large sets of globally distributed non-structured or semi-structured data. The data generated by devices employed in the IMPReSS platform will be produced at very high rates and needs to be pre-processed in a timely manner, in order to be used as input by the data analysis and machine learning modules (as described in Tasks 5.2 and 5.3 of the project proposal).

The current decision greatly affects Work Package 5 (Data Storage, Analysis & Decision Support). This is concerned with providing tools allowing developers to consistently manage massive amount of the acquired data from the smart objects, analyze and extract information and finally transform data into knowledge that is useful for the application domain within the integrated IMPRESS platform. This deliverable, in particular, defines the Data, Policy and Knowledge Storage module specification.

1. IMPReSS Platform Overview

The IMPRESS platform can be seen as a software platform in the cloud designed to manage ubiquitous sensor data. It was engineered to provide a set of tools to help users (end-users and application developers) to build and manage connected smart devices and applications based on connected things.

The platform has been designed to facilitate the integration with sensing technologies, networking applications, data mining and processing tools. This enables users to collect and visualize environmental information while compiling and adding value to such information, in order to generate knowledge about the acquired data.

Figure 1 provides a complete overview of the IMPRESS platform components.

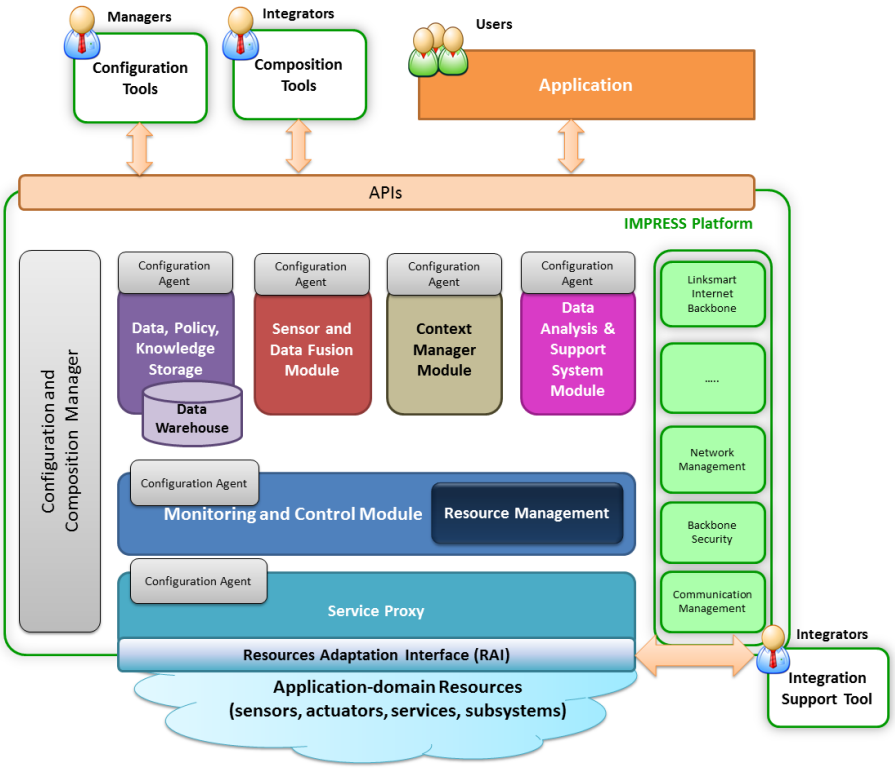

Figure 1: IMPReSS platform Overview.

The Data, Policy and Knowledge Storage module is responsible for managing the persistence of various data and information sets. By leveraging on NoSQL databases, it maintains informations such as historical sensor data, inferred knowledge, policies, configurations and etc. It makes available for others components of the IMPRESS platform services such as the storage of both raw data and enhanced information.

Next sections provide details of data storage specification and implementation.

2. Data, Policy and Knowledge Storage Module

2.1 Data Model

The term data model has been used in the information management community with different meanings and in diverse contexts. In its most general sense, a data model is a concept that describes a collection of conceptual tools for representing real-world entities to be modelled and the relationships among these entities [3].

There are different models that may be used in the data modelling area such as hierarchical, relational, semantic, object-oriented, graph, and semi-structured. Among all of them, the relational model, which introduces the idea of separation between physical and logical levels, is the most popular and widely employed among the business applications [4]. However, this classical model has been criticized for its lack of semantics. Its flat structure imposes difficulties for the user to map the connectivity of the data, both conceptually and during the implementation.

Under the IMPRESS platform, the data semantics and analytics are the fundamental features needed to support the decision making process. Multi sensor data fusion provides a means to fuse raw data into meaningful higher-level information for the users. Moreover, the recognition of the modelled situations requires understanding the technicalities of each sensor, signal processing, sensor fusion techniques to combine readings from different sensors. In such scenario, where the information about the interconnectivity or the topology of the data is more important than, or as important as, the data itself, the data modelling based on graph has several advantages.

First, graphs provide a natural and flexible way to represent information about real world (i.e. real world objects are nodes and relations between different objects are vertexes/edges).

Second, typical graph databases provide built-in structures (i.e. nodes and edges) to represent graphs. Whereas in other databases, relationships between entities in the data model would have to be handled by the modeller at the model level. Or in other words, new tables or columns, at least in the SQL case, would have to be maintained only for the sake of being used as query indirection stages that point to other entities, probably via foreign keys.

For these reasons, the data modelling adopted in the project is based on a property graph representation. In the realm of graphs' morphism, a property graph is a vertex/edge-labeled/attributed, directed, multi-graph. More details, on why a property graph representation was favoured over RDF's edge-labeled directed graphs, will be given in the Architecture section. The data modelling is based on sensor readings arranged in a certain physical environment. The setting may have an infinite number of areas, which in turn may or may not embody other areas within it. Each area may contain an indefinite number of devices that belong to a sensor network. These devices will perform several measurements of the various parameters throughout the day, while it is necessary to store a history of such readings possibly for an indefinite time, depending on application requirements. A generic description of the IMPRESS scenario is shown in Figure 2.

The data modelling has the following types of nodes:

- Area (yellow) - these nodes store a representation of a given monitored environment, such as area type, area name, so on. This type is used to hierarchically divide the environment into classes (e.g. rooms, hall, and garden).
- Device (green) - these nodes represent the devices contained in the environment, such as sensors, actuators, controllers, and mobile devices. This node entails the type of device and its network address.

- Measurement Variable (red) – these nodes represent the variable being measured in the environment, such as humidity, temperature, and energy.
- Measurement History (purple) - the measurement history of each device was modelled in as a linked list, aiming to minimize the impact of query time when historical data grows bigger.
- Category (blue) – A node for classification of devices, e.g. illumination, HVAC, so on. Each category is unique and can classify devices in a one-to-many fashion.

As for the edges, the types are:

- has – these edges link an area to the sub-areas it is composed of. Generating a hierarchy of spatial representations of the measured environment.
- interacts – this edge specifies which measurement variables can be measured by the device it is linked to.
- was measured – this edge link measurement histories, in a chronological order, to the device that measured it. Note that this edge, in particular, contains a timestamp property, representing the time the measurement history it points to was measured.
- comprehends – is the relation between a category and the device it classifies. Each category can comprehend many devices with the same characteristics.

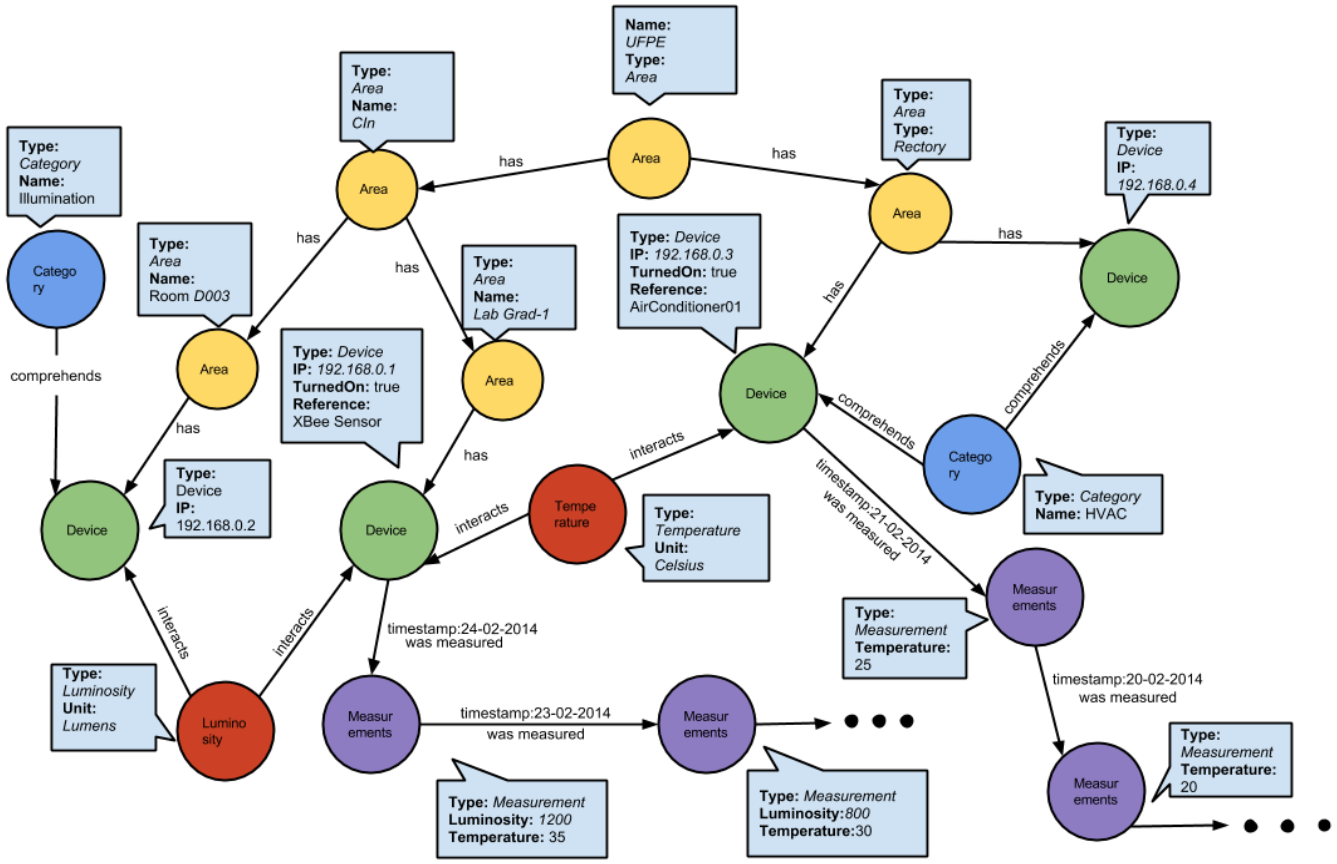

Figure 2: Data Model for the Data, Policy and Knowledge Storage module.

As a result of the data model depicted at Figure 2, flexible and powerful queries could be performed, such as:

- Querying which devices can/cannot measure a given measurement variable. As well as list the areas that have devices measuring their temperature, for instance.
- Querying the area where a given device is located, via the device's IP. Or, alternatively, list all the devices in a given area.
- Querying all the sub-areas of a given area, via its area's name.
- Querying all the measurement histories, in a given time range, for a specific area. Despite the device that measured them.
- Etc.

2.2 Proposed Architecture

The Data and Knowledge Storage module consists of a set of technologies responsible for managing and storing data. These technologies are based on a NoSQL database, more specifically, a graph-based one.

Graph databases are perhaps the most popular graph computing technology. They provide transactional semantics such as ACID, which is typical of local databases, and eventual consistency, which is typical of distributed databases. Different from in-memory graph toolkits, graph databases use the disk to store the graph data. On sufficiently powerful machines, local graph databases can support a couple billion edges while distributed systems can handle hundreds of billions of edges. However most distributed graph-based NoSQL databases, like Neo4j [11], does not provide the means for global graph algorithms to be performed within a reasonable milliseconds time scale, in a hundreds of billions of edges scenario. And since WP5 tasks leverage heavily in the data processing for the machine learning and data fusion techniques, be able to have a continuous feedback loop that works almost in quasi real time and have a global view of the current and past state of the system, mainly due to global graph algorithms, is invaluable.

Considering this practical concern, we adopt the Titan [5] open implementation as distributed graph-based NoSQL database. Therefore, we can represent our data model, as depicted in Figure 2, without any modifications. This data model fits perfectly the knowledge inference case that further WP5 tasks require, since knowledge can be easily represented with graphs as a set of relations between concepts. RDF and ontologies, for instance, are just graphs connecting subjects and objects via a predicate, i.e., triples. Graph-based NoSQL databases, however, have a clear advantage over RDF and ontologies. They have built-in database support to triples, while ontologies and RDF require extra parsers, at the application level, to extract semantics from the employed syntax (e.g. XML, Turtle, Notation 3, etc). Obviously, RDF is a standard and is widely used by the linked data community, however it was not envisioned to be used in a distributed context that suits our proposed use case. It certainly fits well for the use case of the web, with a whole architecture based on documents being exchanged from a web server to clients using a request-reply pattern. But as the size of the a RDF document grows, it is up to the libraries' implementers to figure out how to deal with scalability problems. Like graph partitioning and distributed processing of the RDF documents. Not handling that can hinders the usage of RDF to store vast amounts of data.

As for Titan, graph partitioning, among Titan instances, and distributed batch processing, via Faunus, are already implemented. These two features per se permits Titan to scale horizontally, which was one of our major concerns from the very beginning.

Despite that, property graphs explicitly separate out node/edge specific key/value data from the underlying graph structure as a design-time decision. When using triple stores in practice, most of

the edges turn out to be spurious. Since 'properties' of a node are not first class citizens of the graph structure itself. In RDF, for instance:

:a :hasAge "24".
:a foaf:knows :b.

These are both triples and hence considered graph edges, but only the second one represents connectivity in a graph sense. The first 'edge' is not really an edge, but a property of :a with no meaning outside of :a, since it is simply a literal and not a real entity per se.

As a side note, Titan supports several storage backends. Like Cassandra, which is a column-family NoSQL database developed and open sourced by Facebook in 2008, Hbase [19], which is an open source implementation of Google's BigTable, Oracle Berkeley DB [20] and Akiban Persistit [21]. For this proposed architecture, we favoured Cassandra, due to its maturity and large developer community.

At the scale of hundreds of billions of edges and with several concurrent users, where random access to disk and memory are at play, global graph algorithms are not feasible. What is feasible is local graph algorithms/traversals. Instead of traversing the entire graph, some set of vertices serve as the source (or root) of the traversal. To tackle the need for global graph algorithms/traversals, batch processing graph frameworks can be used. Most of the popular frameworks in this space leverage on Hadoop [16] for storage (HDFS) and processing (MapReduce). These systems are oriented towards global analytics. That is, transversals that pass through the entire graph dataset and, in the case of iterative algorithms, touch the entire graph many times. Such analyses do not run in real-time. However, because they perform global scans of the data, they can leverage sequential reads from disk (see [6]).

Along with Titan, we use Faunus [7] for distributed batch processing and support graph analytics in a timely manner. Faunus is able to distribute queries steps in the available Titan clusters and load balancing the workload. Therefore, drastically reducing latency for database operations in graphs with billions of edges and nodes. Faunus works on top of Hadoop, which is an open source project backed by the Apache Foundation and based on Google's Map-Reduce white paper. Also, it is noteworthy that both Titan and Faunus, following the trend of technologies around NoSQL databases, can scale horizontally by adding more clusters. That is, if in a given scenario the current servers can no longer handle the load, one may simply execute more instances of Faunus and Titan in a divide and conquer strategy to attend the usual batch of database queries performed in our proposed architecture.

For querying, we use a domain-specific language, the Gremlim language [8], which can perform complex operations in multi-relational graphs, called property graphs. Gremlim is based on the Groovy language [10]. With Gremlin it is possible to perform operations such as the addition or removal of nodes/edges, manipulate the graph indexes, complex graphs transversals, etc. Also, it is part of the Blueprints [5] stack. The equivalent of Gremlin in the RDF world is SPARQL [17]. Comparatively, SPARQL does not support iteration/looping, consequently being particularly hard to compute graph-structural metrics like centrality. In that sense, Gremlin is more powerful than SPARQL.

The Blueprints stack is an open source property graph model for a common interface that can facilitate the interaction with the underlying supported graph databases. Among the supported

databases there are: Neo4j [11], Titan [5], OrientDB [12], SparkSee [13] and more. More importantly, these cited databases are currently seen as some of the major players in terms of graph database usage. Despite the considerable number of supported alternatives, leveraging on a common interface can help us to avoid having the IMPRESS architecture tied to specific proprietary solutions. Such problem could impose barriers in the event of changing the underlying graph database used by the Impress cloud. The blueprints stack is maintained by a group called Tinkerpop [14], which entails as one of its members the lead developer of Titan. Also, notice that Blueprints is not a programming library per se. It is an API that has several implementations in many programming languages, like Java and Python.

Finally, we chose to use a Rexter server [9], which is also part of the Blueprints stack, to be the interface exposed for developers to execute database operations, via Gremlim queries. Or, in other words, the Rexter server allows developers to communicate with Blueprints-enabled graphs in a language agnostic fashion. That is, we could change the underlying NoSQL graph database at any time, without requiring any source code change in the clients of the Data, Policy and Knowledge Storage module. Also, Blueprints is a Java API for graph databases. So, by using Rexter, developers can access the Blueprints API over HTTP/REST directly or by using libraries that support Blueprints API, like PyBulbs [18] for Python. Both Titan and Faunus clusters are just part of the required infrastructure, but they are not directly exposed to other modules. The Rexter server supports both a JSON-based REST interface and a binary protocol called RexPro. In our architecture, we favoured the RexPro case due to its smaller footprint. When a Gremlim query is received, the Rexter server passes it to one of the Faunus clusters and waits for the response, which is then replied back to the requester.

Finally, Figure 3 shows the overview of the Data, Policy and Knowledge Storage module architecture.

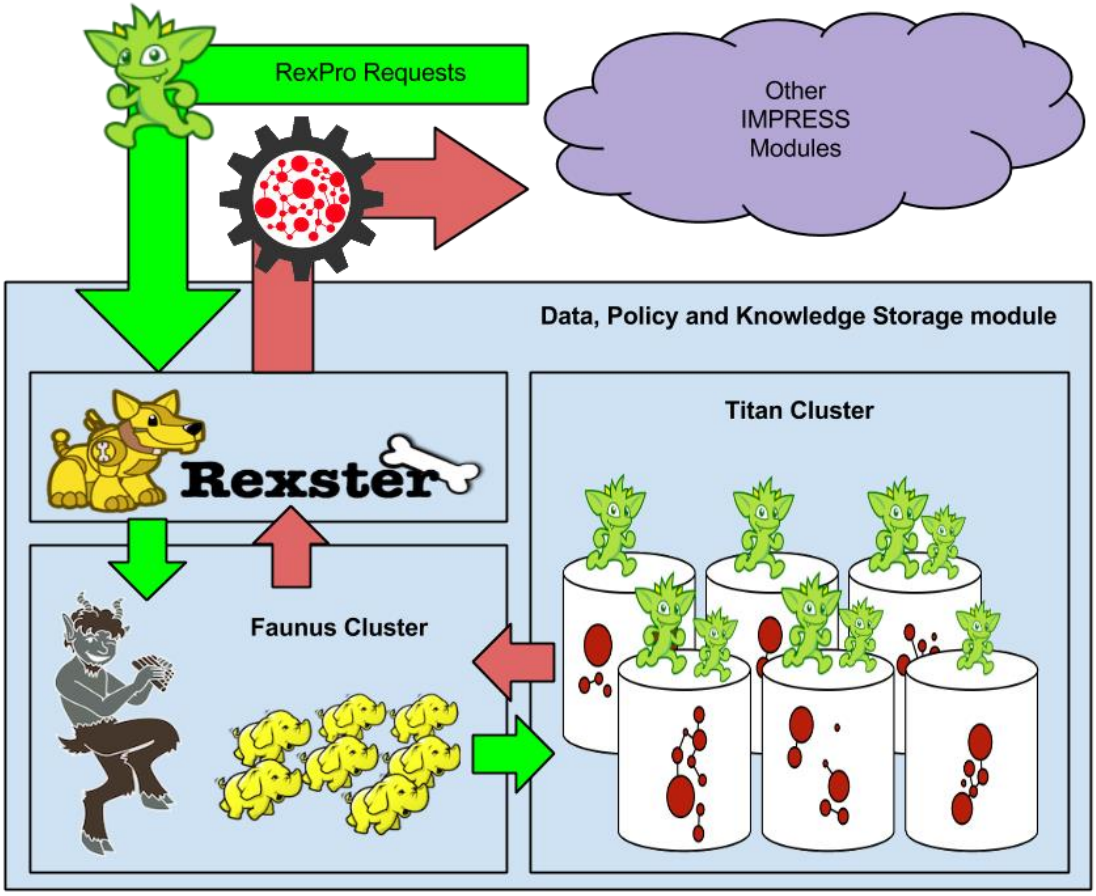

Figure 3: Data, Policy and Knowledge Storage module architecture.

3. Domain Specific Language – DSL

Gremlim is a flexible and powerful query language, but it certainly requires previous knowledge about graphs structures (i.e. nodes and edges) and their algorithms. Albeit graphs are certainly not a new concept, especially due to their mathematical origin, their usage for modelling data and consequently the awareness of their implications still not mainstream. Therefore, we have built a domain specific language (DSL) on the top of Gremlim, so that we can abstract the most common operations technicalities (e.g. adding/removing nodes, dealing with relationship between nodes, etc) regarding the data model depicted in Figure 2. Like, for instance, instead of directly creating a node for a device, set its IP property, link it to the intended area and to a set of measurement variable nodes, a developer could use the createDevice (ip,area_contained,measure_types) construct that is part of the DSL, so that most part of the graph manipulation process regarding the data model is hidden from clients. Alternatively, the DSL can be seen as a higher level interface (API) to interact with the proposed data model depicted in Figure 2. From this point on, we will refer to this new DSL as Impress' DSL. The Impress' DSL, however, is not as flexible as Gremlim's core constructs. Nevertheless, since Impress' DSL is basically just another layer on top of Gremlim, the underlying query language remains Gremlim. And that is an important decision choice we made, since it can please both newcomers, eager to perform simple queries using Impress' DSL, and more advanced users, that can mix Gremlim core constructs with Impress DSL's calls.

3.1 Impress' DSL Documentation

area(name)
Arguments:
name: String , null

Return:
Returns an area if a name is specified or a list of all areas if no name is passed as argument.

Definition:
Gremlin.defineStep('area',[Vertex,Pipe],
{name -> _().ifThenElse{name == null}
{it.has('Type','Area')}
{it.has('Type','Area').has('Name',name)}})

Example for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.area()**

```
{

   "results": [
      {
         "Name": "Lab Grad-1",
         "Type": "Area",
         "_id": 31744,
         "_type": "vertex"
      },
      {
```

```
            "Name": "UFPE",
            "OptionalParameters": {
               "Description": "University",
               "UF": "Pernambuco"
            },
            "Type": "Area",
            "_id": 30208,
            "_type": "vertex"
         },
         {
            "Name": "Theater UFPE",
            "OptionalParameters": {
               "Description": "Facility where the UFPE Demo takes place"
            },
            "Type": "Area",
            "_id": 30720,
            "_type": "vertex"
         },

          [...]
      ],
      "success": true,
      "version": "2.5.0",
      "queryTime": 908.6409

   }

   http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=g.V.area().map()
   {
      "results": [

         {
            "Name": "Lab Grad-1",
            "Type": "Area"
         },
         {
            "Name": "UFPE",
            "OptionalParameters": {
               "Description": "University",
               "UF": "Pernambuco"
            },
            "Type": "Area"
         },
         {
            "Name": "Theater UFPE",
            "OptionalParameters": {
               "Description": "Facility where the UFPE Demo takes place"
            },
            "Type": "Area"
         },
```

```
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 890.9162

}
```


http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.area().next()**


```
{
    "results": [
        {
            "Name": "Lab Grad-1",
            "Type": "Area",
            "_id": 31744,
            "_type": "vertex"
        },
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 908.6409

}
```


http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.area("UFPE")**


```
{
    "results": [
        {
            "Name": "UFPE",
            "OptionalParameters": {
                "Description": "University",
                "UF": "Pernambuco"
            },
            "Type": "Area",
            "_id": 30208,
            "_type": "vertex"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 1319.4897
}
```


areaPerDevice(ip)

Arguments:
ip: String

Return:
Return an area containing the device specified

Definition:
Gremlin.defineStep('areaPerDevice',[Vertex,Pipe], {ip -> _().ifThenElse{ip == null}{it.has('Type','Device').has('IP',it.IP).in('has')}{it.has('Type','Device').has('IP',ip).in('has')}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.areaPerDevice("192.168.0.1")**


```
{
   "results": [
      {
         "Name": "Lab Grad-1",
         "Type": "Area",
         "_id": 31744,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 1307.5148
}
```


areaPerUnit(unit)

Arguments:
unit: String

Return:
Returns a list of area that contains the unit specified measured by at least one device.

Definition:
Gremlin.defineStep('areaPerDevice',[Vertex,Pipe],
{ip -> _().ifThenElse{ip == null}
{it.has('Type','Device').has('IP',it.IP).in('has')}
{it.has('Type','Device').has('IP',ip).in('has')}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.areaPerUnit("Temperature")**


```
{
   "results": [
      {
         "Name": "Lab Grad-1",
         "Type": "Area",
         "_id": 31744,
         "_type": "vertex"
      },
      {
         "Name": "Rectory",
         "Type": "Area",
```

```
            "_id": 30976,
            "_type": "vertex"
         }
      ],
      "success": true,
      "version": "2.5.0",
      "queryTime": 1247.4387

   }
```


device(ip)

Arguments:
ip: String, null

Return:
Return a device if the ip is specified or return a list of all devices if none ip is passed.

Definition:
Gremlin.defineStep('device',[Vertex,Pipe],
{ip -> _().ifThenElse{ip == null}
{it.has('Type','Device')}
{it.has('Type','Device').has('IP',ip)}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.device("192.168.0.1")**


```
{
   "results": [
      {
         "OptionalParameters": {
            "Reference": "XBee Sensor"
         },
         "Type": "Device",
         "IP": "192.168.0.1",
         "_id": 33792,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 985.8967

}
```


devicePerArea(name)

Arguments:
name: String

Return:

Return a list of devices which is in the area.

Definition:
Gremlin.defineStep('devicePerArea',[Vertex,Pipe],
{name ->_().ifThenElse{name == null}
{it.has('Type','Area').has('Name',it.Name).out.has('Type','Device')}
{it.has('Type','Area').has('Name',name).as('x').out().loop('x')
{it.object.Type != "Device"}.map}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.devicePerArea("Theater UFPE")**


```
{
   "results": [
      {
         "OptionalParameters": {
            "Reference": "XBee Sensor"
         },
         "Type": "Device",
         "IP": "192.168.0.6"
      },
      {
         "OptionalParameters": {
            "Reference": "SmartPlug"
         },
         "Type": "Device",
         "IP": "192.168.0.7"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 1288.0328
}
```


devicePerUnit(type)

Arguments:
type: String

Return:
Return a list of devices which are measuring the unit passed

Definition:
Gremlin.defineStep('devicePerUnit',[Vertex,Pipe],
{unit -> _().ifThenElse{unit == null}
{it.has('Type',it.Type).out}
{it.has('Type',unit).out}})

Examples for a graph *g*:

http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.devicePerUnit("Temperature")**


```
{
   "results": [
      {
         "OptionalParameters": {
            "Reference": "XBee Sensor"
         },
         "Type": "Device",
         "IP": "192.168.0.1",
         "_id": 33792,
         "_type": "vertex"
      },
      {
         "OptionalParameters": {
            "Reference": "AirConditioner01"
         },
         "Type": "Device",
         "IP": "192.168.0.3",
         "_id": 34304,
         "_type": "vertex"
      },
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 1303.297
}
```


measurementFromArea(name)

Arguments:
name: String, null

Return:
The return is all measurements from that area

Definition:
Gremlin.defineStep('measurementFromArea',[Vertex,Pipe],
{name ->_().ifThenElse{name == null}
{it.has('Type','Area').has('Name',it.Name).out.has('Type','Device'). out('was measured')}
{it.has('Type','Area').has('Name',name).out.has('Type','Device').out('was measured')}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.measurementFromArea("Theater UFPE")**


```
{
   "results": [
      {
         "Type": "Measurement",
         "Power": 110,
```

```
            "Humidity": 22,
            "_id": 39936,
            "_type": "vertex"
         }
      ],
      "success": true,
      "version": "2.5.0",
      "queryTime": 910.292201
   }
```


measurementFromDevice(ip)

Arguments:
ip: String

Return:
The return is the last measurement from that ip

Definition:
Gremlin.defineStep('measurementFromDevice',[Vertex,Pipe],
{ip ->_().ifThenElse{ip == null}
{it.has('Type','Device').has('IP',it.IP).out('was measured')}
{it.has('Type','Device').has('IP',ip) .out('was measured')}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.measurementFromDevice("192.168.0.1")**


```
{
   "results": [
      {
         "Type": "Measurement",
         "Temperature": 10,
         "Luminosity": 1290,
         "_id": 38912,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 954.5627
}
```


// Example where you take the in going edge of the last measurement
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.measurementFromDevice("192.168.0.1").inE()**


```
{
   "results": [
      {
         "timestamp": "Mon Nov 10 18:29:05 UTC 2014",
```

```
            "_id": "nnk-q2o-hed-u0w",
            "_type": "edge",
            "_outV": 33792,
            "_inV": 38912,
            "_label": "was measured"
         }
      ],
      "success": true,
      "version": "2.5.0",
      "queryTime": 1336.9622
   }
```


measurementFromUnit(type)

Arguments:
type: String

Return:
The return is all measurements of that unit type

Definition:
Gremlin.defineStep('measurementFromUnit',[Vertex,Pipe],
{type ->_().ifThenElse{type == null}
{it.has('Type',it.Type).devicePerUnit.measurementFromDevice}
{it.has('Type',type).devicePerUnit.measurementFromDevice}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.measurementFromUnit("Temperature")**


```
{
   "results": [
      {
         "Type": "Measurement",
         "Temperature": 10,
         "Luminosity": 1290,
         "_id": 38912,
         "_type": "vertex"
      },
      {
         "Type": "Measurement",
         "Temperature": 25,
         "_id": 39424,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 1316.4512
}
```


unitPerDevice(ip)

Arguments:
ip: String

Return:
Show all units measured by the device specified on ip.

Definition:
Gremlin.defineStep('unitPerDevice',[Vertex,Pipe],
{ip -> _().ifThenElse{ip == null}
{it.has('Type','Device').has('IP',it.IP).in('interacts')}
{it.has('Type','Device').has('IP',ip).in('interacts')}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.unitPerDevice("192.168.0.1")**


```
{
   "results": [
      {
         "Type": "Temperature",
         "Unit": "Celsius",
         "_id": 32000,
         "_type": "vertex"
      },
      {
         "Type": "Luminosity",
         "Unit": "Lumens",
         "_id": 32768,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 1386.8027
}
```


unitPerArea(name)

Arguments:
name: String

Return:
Show all units measured on the area which name is given.

Definition:
Gremlin.defineStep('unitPerArea',[Vertex,Pipe],
{name -> _().ifThenElse{name == null}
{it.has('Type','Area').has('Name',it.Name).devicePerArea.unitPerDevice.unique()}
{it.has('Type','Area').has('Name',name).devicePerArea.unitPerDevice.unique()}})

Examples for a graph *g*:

http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.unitPerArea("Theater UFPE")**


```
{
   "results": [
      {
         "Type": "Humidity",
         "Unit": "UR",
         "_id": 32256,
         "_type": "vertex"
      },
      {
         "Type": "Power",
         "Unit": "Watts",
         "_id": 32512,
         "_type": "vertex"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 1236.0896
}
```


setOptionalParameters(parameters)

Arguments:
parameters: Map

Return:
Adds optional parameters to a vertex. Can be used combined with "area" and "device" steps.

Definition:
Gremlin.defineStep('setOptionalParameters',[Vertex,Pipe],
{parameters -> _().ifThenElse{it.OptionalParameters != null}
{it.OptionalParameters += parameters; g.commit()}
{it.OptionalParameters = parameters; g.commit()}})

Examples for a graph *g*:
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**g.V.area("Lab Grad-1").setOptionalParameters(["Description":"CS Laboratory"])**

createDevice(deviceIp, areaContained, category, measurementTypes, optionalParameters)

Arguments:
deviceIp, areaContained, category: String
measurementTypes: List<type>
optionalParameters: Map<type>

Return:
True, if the device was sucessfull added, or False if an error ocurred.

Definition:

```
def createDevice(deviceIp, areaContained, category, measurementTypes, optionalParameters=[]){
    g = rexster.getGraph("graph");

    devProps = [Type:'Device'];
    devProps['IP'] = deviceIp;

    device = g.addVertex(devProps);

    for(type in measurementTypes){
        vertex = g.V.has("Type",type).next();
        g.addEdge(vertex,device,'interacts');
    }
    if(optionalParameters){
        device["OptionalParameters"] = optionalParameters;
    }

    categoryVertex = g.V.category(category).next();
    g.addEdge(categoryVertex,device,"comprehends");

    area = g.V.area(areaContained).next();
    g.addEdge(area, device, 'has');
    g.commit();
}
```

Examples:

http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=**createDevice("192.168.0.1","Lab Grad-1","Sensoring",["Temperature","Luminosity"],["Reference":"XBee Sensor"])**

createArea(name, areaContained, optionalParameters)

Arguments:

name, areaContained: String

aditionalParameters: Map <type>

Return:

True, if it was successfully added, or False if an error occurred.

Definition:

```
def createArea(name, areaContained=""){
    g = rexster.getGraph("graph");
    area = g.addVertex([Type:'Area',Name:name]);

    if(areaContained.length() > 0){
        g.addEdge(area, g.V.area(areaContained).next(), "has");
    }

    g.commit();
```

```
}

def createArea(name, String areaContained, optionalParameters=[]){
    g = rexster.getGraph("graph");
    area = g.addVertex([Type:'Area',Name:name]);

    if(areaContained.length() > 0){
        g.addEdge(area, g.V.area(areaContained).next(), "has");
    }
    if(optionalParameters){
        area["OptionalParameters"] = optionalParameters;
    }
    g.commit();
}

def createArea(name, Map optionalParameters){
    g = rexster.getGraph("graph");
    area = g.addVertex([Type:'Area',Name:name]);

    if(optionalParameters){
        area["OptionalParameters"] = optionalParameters;
    }
    g.commit();
}
```

Examples:

// A root area, with an optional parameter.

http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=

createArea("UFPE", ["Description":"University", "UF":"Pernambuco"])

// An area named "CIN" contained in area "UFPE"

http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=

createArea("CIN", "UFPE")

// An area named "Theater UFPE", contained in area "UFPE", with an optional parameter

http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script= **createArea("Theater UFPE", "UFPE" ,["Description":"Facility where the UFPE Demo takes place"])**

// just a root area named Room D003

http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=

createArea("Room D003")

measurementsPerTicksAndTimestamp(deviceNetworkId, numTicks, beginTimestamp, endTimestamp)

Arguments:

deviceNetworkId: String

numTicks: Integer

Return:

Returns a set of means of measurements values between two dates given (beginTimestamp and endTimestamp). The number of means is defined by numTicks, i.e, the number of fusions of measurements values in the timestamps interval.

Definition:

```
def measurementsPerTicksAndTimestamp(deviceNetworkId, numTicks, beginTimestamp, endTimestamp){
    g = rexster.getGraph("graph");

    firstVertex =  g.V.measurementFromDevice(deviceNetworkId)
                .as('x').out("was measured")
                .loop('x'){
                    new Date(it.object.outE.map.next()
                    .timestamp.toString())>=(new Date(endTimestamp))
                }.next()

    path =  firstVertex.as('x').out("was measured")
        .loop('x'){                                    t
            new Date(it.object.outE.map.next()
            .timestamp.toString())>=(new Date(beginTimestamp))
        }.path.next();

    pathList=[];
    pathJumps=numTicks;

    numVertices = 0;
    for(vertex in path){
        numVertices += 1;
    }
    numSamples = (numVertices/numTicks).toInteger();
    pathList=[];

    newMeasurement = firstVertex.map.next();
    newMeasurement = resetMeasurement(newMeasurement);

    node=0;
    for(i=0; i<numTicks ;i++){
        for(j=0; j<numSamples ;j++){
            newMeasurement = incMeasurement(newMeasurement,path[node]);
            node++;
        }
        newMeasurement = meanMeasurement(newMeasurement,numSamples);
        pathList+=newMeasurement.clone()
        newMeasurement = resetMeasurement(newMeasurement);
    }
    return pathList
}
```

Examples for a graph *g*:

// between beginTimestamp and endTimestamp there are 4 measurements

// and numTicks is 4, so 4 unmodified measurements are returned
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script=
measurementsPerTicksAndTimestamp("192.168.0.1",4,
"Mon Nov 10 18:28:57 UTC 2014",
"Mon Nov 10 18:29:02 UTC 2014")


```
{

   "results": [
      {
         "Luminosity": 1270,
         "Temperature": 13,
         "Type": "Measurement"
      },
      {
         "Luminosity": 1270,
         "Temperature": 15,
         "Type": "Measurement"
      },
      {
         "Luminosity": 1270,
         "Temperature": 17,
         "Type": "Measurement"
      },
      {
         "Luminosity": 1270,
         "Temperature": 19,
         "Type": "Measurement"
      }
   ],
   "success": true,
   "version": "2.5.0",
   "queryTime": 535.4372

}
```


// Now numTicks is 2, so two measurements will be returned, the first with the
// mean of the two first measurements, and the second with the mean of
// measurements 3 and 4.
http://impress-storage-ip:8182/graphs/graph/tp/gremlin?script
=**measurementsPerTicksAndTimestamp("192.168.0.1",2,**
"Mon Nov 10 18:28:57 UTC 2014",
"Mon Nov 10 18:29:02 UTC 2014")


```
{
   "results": [
      {
         "Luminosity": 1270,
         "Temperature": 14,
         "Type": "Measurement"
      },
      {
```

```
            "Luminosity": 1270,
            "Temperature": 18,
            "Type": "Measurement"
        }
    ],
    "success": true,
    "version": "2.5.0",
    "queryTime": 511.6438
}
```


3.1.1 Impress' DSL Steps

Steps can be used in two different notations:

Postfix notation:
g.V.data.step

InFix notation:
g.V.step(data)

The returned value is the same in both cases. Therefore, a user can aggregate several steps in a single Gremlim query.

For example:
g.V.data.step1.step2(somedata).step3.step4……

The "g.V" indicates that all the database, after further composition of steps the set of data will be restricted due to previous steps.

Examples:

g.V -> All database
g.V.step1 -> Sub-set of database after step1
g.V.step1.step2 -> Sub-set of databse after step1 and step2

Step	Description	Return
area (String name)	Returns an area, if a name is specified, or a list of all areas, if no name is passed as argument.	A set of areas.
areaPerDevice (String ip)	Returns an area containing the device with the specified IP.	A set of areas.
areaPerUnit (String unit)	Returns a list of area that contains the unit specified measured by at least one device.	A set of areas.
areaFromArea (String name)	Returns a list of areas belonging to an specific area.	A set of areas.
category (String name)	Returns a category, if a name is specified, or a list of all devices, if no name is passed as argument.	A set of categories.
device (String ip)	Returns a device, if an IP is specified, or a list of all devices, if no IP is passed as argument.	A set of devices.
devicePerArea (String name)	Return a list of devices which is in the area.	A set of devices.
devicePerUnit (String type)	Returns a list of devices which are measuring the specified unit (e.g. temperature, light intensity and etc).	A set of devices.
devicePerCategory (String name)	Returns a list of device with a category in comnext(e.g. illumination, HVAC).	A set of devices.
measurementFromArea (String name)	Returns all measurements made by devices in the specified area.	A set of measurements.
measurementFromDevice (String ip)	Returns all measurements from a device with the specified IP.	A set of measurements.
measurementFromUnit (String type)	Returns all measurements containing the specified unit type (e.g. temperature, light intensity and etc).	A set of measurements.
unitPerDevice (String ip)	Show all units (e.g. temperature, light intensity and etc) measured by the device with the specified IP.	A set of units.
unitPerArea (String name)	Returns all units (e.g. temperature, light intensity and etc) measured on the specified area.	A set of units.
setOptionalParameters (Map parameters)	Add optional parameters to an area or a	Null.

	device. A map of key(attribute name)=value(attribute value)	

3.1.2 Gremlim's Useful Steps

The steps below, are part of Gremlim's core steps, but we decided to cite them due to their usefulness.

Step	Description	Return
map	Show all the property and values from a node. Example: data.map	A description of the node.
next()	Get the next element from a set. Example: set.next().next().next() Will return the 3rd element from set.	The next node of the container.
property*	Access the property value from some node or edge. Example: data.property* Will show the property* from the data node. data.property* = new_value Will update property* with new_value..	Null.
remove()	Remove an element from the database. Example: data.remove() Will remove data node.	Null.

3.1.3 Impress' DSL Functions

Different from steps, functions cannot be aggregated in a single call. In other words, prefix and postfix notations cannot be used.

For example:
function(parameter1, parameter2…)

Function	Description	Result
createDevice (String deviceIp, String areaContained, List<String> measurementTypes, Map optional Parameters)	Creates and saves, on the database, a new device, an edge between the new device and its area, the types of measurements it can perform and optional parameters for unique.	The representation of a edge, if the device was successfully created, or an error message.
createArea (String name, String areaContained, Map optionalParameters)	Creates and saves, on the database, a new area, an edge between the new area and area which it is part of. Also additional parameters for specific area characteristics.	The representation of a edge, if the device was successfully created, or an error message.
createCategory (String name)	Creates and saves a category.	A vertex with type category to classify devices.
createMeasurement(deviceIp, values)	Creates a measurement vertex for a device identified by deviceIp with values determined by the values parameter.	A vertex containing values passed as parameters with an edge from the device identified by deviceIp and an edge to the previous measurement.
createMeasurementVariable (variableName, unitName)	Creates a new vertex of measurement variable type. When a device is created with an specific measurement type, the measurement type vertex created with the same name is linked to this device.	A vertex with a measurement type name, e.g. temperature, and measurement unit name, e.g. Celsius.
measurementsPerTicks (deviceNetworkId, numTicks)	Returns the last measurements given the number of last measurements (numTicks).	A set of measurements. Measurement being a Map of measurement types and measurement values.
measurementsPerTicksAndTimestamp (deviceNetworkId, numTicks, beginTimestamp, endTimestamp)	Returns a set of means of measurements values between two dates given (beginTimestamp and endTimestamp). The number of means is defined by numTicks, i.e, the number of fusions of measurements values in the timestamps interval.	A set of means of measurements values. Measurement being a Map of measurement types and measurement values.

4. Initial Performance Evaluation

To evaluate the performance of our proposed architecture, we populated an instance of Titan with the data model proposed in Figure 2, via a Rexter Server. This is important since this evaluation can be envisaged as a real performance evaluation of the proposed architecture, not just an indiscriminate stress test. Especially, because query times depends on the data model used. We conducted the initial experiments using the following number of nodes:

- Devices: 30
- Measurement Histories per device: 500
- Areas: 7
- Measurement Variables: 4 (i.e. energy consumption, temperature, humidity and light intensity)

As a result, this experiment generated a total of 120.252 vertices and 120.966 edges in our Titan instance. Given the populated database, we performed 10,000 random automated queries, in order to evaluate query performance for our model. Both the populate and query performance steps were executed by two Python scripts, developed for this matter. These scripts were executed on an Intel Core i3 - 2100 CPU @ 3.10 GHz with two GB of Ram. The query performance benchmark took 200 seconds to finish, with an average of 20 milliseconds per query. It is also noteworthy that 75% of the queries run in less than this average time, as depicted in Figure 4.

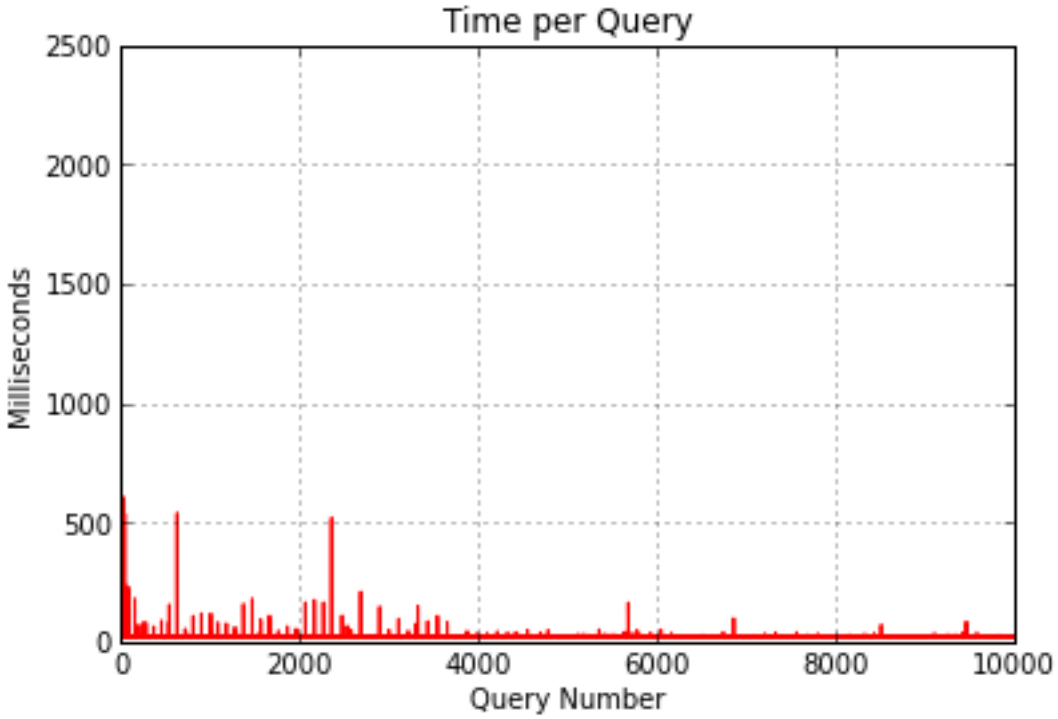

Figure 4: Histogram of query time per query number.

In the end, as previously said, the performance benchmark had only one instance of Titan as part of it. Obviously, in real case scenarios, that would hardly be the case. The major advantage of NoSQL technologies, in general, is exactly the capability of distributing the workload with all the servers running a database instance.

References

[1] Jayavardhana G., Rajkumar B., Slaven M., Marimuthu P. Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Comp. Syst. 29(7): 1645-1660 (2013).

[2] IMPRESS (Intelligent System Development Platform for Intelligent and Sustainable Society) project, Description of Work. EU- Brazil research and development Cooperation

[3] Silberschartz, A., Korth, H.F., and Sudarshan, S. Data models. ACM Computing Surveys, 28, 1, 105-108.

[4] Levene, M., Poulovassilis, A. The hypernode model and its associated query language, Proceedings of the fifth Jerusalem conference on Information technology, p.520-530, 1990.

[5] Titan – Distributed Graph Database. http://thinkaurelius.github.io/titan/

[6] Adam Jacobs, The Pathologies of Big Data, 1010data Inc., 2009.
https://queue.acm.org/detail.cfm?id=1563874

[7] Faunus – Graph Analytics Engine. http://thinkaurelius.github.io/faunus/

[8] Gremlin – Graph Database Language. https://github.com/tinkerpop/gremlin

[9] Rexster Graph Server. https://github.com/thinkaurelius/titan/wiki/Rexster-Graph-Server

[10] Groovy. http://groovy.codehaus.org/

[11] Neo4J. http://www.neo4j.org/

[12] OrientDB. http://www.orientechnologies.com/orientdb/

[13] SparkSee. http://www.sparsity-technologies.com/

[14] Tinkerpop. http://www.tinkerpop.com/

[15] Blueprints. https://github.com/tinkerpop/blueprints/wiki

[16] Hadoop. https://hadoop.apache.org/

[17] SPARQL. http://www.w3.org/TR/rdf-sparql-query/

[18] PyBulbs. http://bulbflow.com/overview/

[19] Hbase. https://hbase.apache.org/

[20] Oracle Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

[21] Akiban Persistit. http://www.akiban.com/