Title:	*Data synchronisation layer*
Authors:	*Marco Scavuzzo (POLIMI), Elisabetta Di Nitto (POLIMI), Jacek Dominiak (CA)*
Editor:	*Elisabetta Di Nitto (POLIMI)*
Reviewers:	*Marcos ALMEIDA (Softeam), Peter MATTHEWS (CA)*
Identifier:	*Deliverable # D6.7*
Nature:	*Prototype*
Version:	*1.0*
Date:	*02/04/2015*
Status:	*Final*
Diss. level:	*Public*

Executive Summary

In this deliverable we release the MODAClouds components to support migration within two important classes of NoSQL, that is, graph and columnar. This document accompanies the software components and supports the reader in the steps toward their installation and usage.

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 318484 (MODAClouds).

Members of the MODAClouds consortium:

Politecnico di Milano	Italy
StiftelsenSintef	Norway
Institutul E-Austria Timisoara	Romania
Imperial College of Science, Technology and Medicine	United Kingdom
SOFTEAM	France
Siemens srl	Romania
BOC Information Systems GMBH	Austria
Flexiant Limited	United Kingdom
ATOS Spain S.A.	Spain
CA Technologies Development Spain S.A.	Spain

Published MODAClouds documents

These documents are all available from the project website located at http://www.modaclouds.eu/

Contents

1. Introduction

1.1 Context and objectives

MODAClouds supports migration of applications from a cloud to another to ensure high availability and ability to react to a sudden decrease of Quality of Service (QoS) in the original cloud. If the application is exploiting some Data Base Management System (DBMS) installed or made available as a service in the original cloud, an important issue to be considered during migration is the transfer of data from this DBMS into a new one, typically installed or made available into the target cloud (see D3.1.4).
While commercial migration tools are available for relational DBMS, the situation of NoSQL is still "greenfield". In this context, DBMS often offer an incompatible data model that makes migration a difficult task.
In this deliverable we release the MODAClouds components to support migration within two important classes of NoSQL, that is, graph and columnar. Graph databases are very useful when we need to keep track of the relationships between various concepts and we need to be able to browse the database by navigating such relationships. Columnar databases have been developed several years ago to support data analytics and have been adopted and transformed in the NoSQL context to handle very large quantities of data by distributing them in different nodes.
The component to support migration between graph databases has been developing by relying on the featured offered by "blueprints" (https://github.com/tinkerpop/blueprints/). This component supports the connection and interaction with graph databases. The resulting migration component has been validated by migrating the data stored in Venues 4Clouds among three different graph databases, ArangoDB (https://www.arangodb.com), Neo4j (http://neo4j.com) and Titan (http://thinkaurelius.github.io/titan/).
The component to support migration between columnar databases is called Hegira4Clouds and supports also synchronization of different replicas. Thanks to this feature, the application can continue operating on data even during migration, as the actions it performs are propagated to all replicas in a seamless way. While the correctness and performance of the migration approach has been carefully validated as reported in D6.6, at the time of writing the synchronization part has been just completed and will be validated in the next months leading to further documentation that will be linked from the tool repository on github (http://deib-polimi.github.io/hegira-api/).

Within the context of the MODAClouds platform, the process of data migration (and synchronization in the case of columnar databases) can be started either manually by a system operator, or automatically by Models@Runtime, as part of an adaptation policy (see D6.5.3). The integration issues to be tackled are described in D3.4.1. A complete development of such integration will be the focus of this last part of the project. The main issue to be considered is time. The migration process typically takes a significant amount of time, which increases with the size of the dataset to be migrated. This means that data migration cannot be activated at the time the application should be moved to a new cloud, but the need for migration has to be foreseen well in advance and data migration has to be performed as a background activity before the application actually starts its execution in the new cloud.

1.2 Structure of the document

This document accompanies the software components and supports the reader in the steps toward their installation and usage. At first, the document presents an overview of the classes of databases supported for data migration and describes the general approach adopted by the components to actually perform data migration (Section 2 and Section 3). Section 4 resumes the main achievements. Finally, the appendices provide details on the achievements and point to the tool repositories and wikis for further, up-to-date information. In particular, the component for migrating graph databases is presented in Appendix A, while Hegira4Clouds is presented in Appendix B. Both appendixes are organized as follows:

- They describe the APIs of each component (Sections A.1 and B.1).
- They provide guidelines for installation (Sections A.2 and B.2) and usage of the components (Sections A.3 and B.3), by describing also an example of migration and discussing on how to extend the migration components to incorporate new databases.

2 Data migration for graph NoSQL databases

2.1 Overview on graph databases

Over the last few years we see a paradigm shift in the way data is consumed. With the ever-increasing need to store but not always consume, different types of data, a new approach has been developed in order to accommodate new type of irregular data. This approach to the data store, commonly known as Not Only SQL or NoSQL, has allowed storing schema-less data, which can be easily scaled in a not replicable but rather concurrent fashion. One type of such databases is graph databases.
Graph databases allow you to store entities, commonly referred to as nodes, and relationships between these entities called edges. Node, in an essence, is an instance of an object in the application with specific properties. Edges, which also have properties, but with reinforced "from node" and "to node" properties to describe node references have directional significance. Nodes are organized by relationships, which allow you to find interesting patterns between the nodes. The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships and it's properties.
Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship ("who is my manager" is a common example). In relational databases we model the graph beforehand based on the Traversal we want. If the Traversal changes, the data will have to change.
In graph databases, traversing the joins or relationships is very fast. Traversing persisted relationships is generally faster than calculating them for every query.
Nodes can have different types of relationships between them, allowing you to both represent relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access.
Relationships are first-class citizens in graph databases. Most of the value of graph databases is derived from the relationships. Using edges properties, we can add intelligence to the relationship—for example, what is the distance between the nodes, or what aspects are shared between the nodes as if modelled from the graph perspective, the search can begin from the nodes inner relation level rather than by analysing all the document properties.
There are more and more graph databases available, such as Neo4J and Titan. The need for a graph-based approach is now recognized even by the traditional vendors that are incorporating such feature in their main offer. The most prominent example is Oracle with its Oracle Spatial and Graph implementation.

2.2 Short description of the migration approach

The approach for the data migration within the graph databases is a fairly complex task as the technology of graph databases is not a mature subject; therefore one commonly accepted standard for data representation has yet to be emerged. There are however libraries, which provide abstraction layer on top of the data storage which provide the common mechanisms and interfaces to store and retrieve the dataset without consideration of underlying data storage system. One of the most common set of tools that allow for the common communication interface is "blueprints" by thinkerpop[1].
Blueprints is a collection of interfaces, implementations, *ouplementations*, and test suites for the property graph data model. An ouplementation[2] is a software layer that transforms a graph-based model into a different format Blueprints is analogous to the JDBC, but for graph databases. As such, it provides a common set of interfaces to allow developers to plug-and-play their graph database

[1] https://github.com/tinkerpop/blueprints/

[2] http://www.tinkerpop.com/docs/wikidocs/blueprints/2.1.0/Ouplementations.html

backend[3]. Using blueprints, we were able to extract the data from the graph databases and accommodate transformation mechanisms in order to save the data to the new set of graph technologies.
The data migration tool set allows predefining the schema with searchable attributes, which can be used in order to transform or rebuild the graph in the new data schema, all of which are in detail described within the next chapter.

[3] https://github.com/tinkerpop/blueprints/wiki

3 Data migration and synchronization for column-based NoSQL databases

3.1 Overview on column-based NoSQL databases

Column-based NoSQL databases owe their name to the data model proposed by Google BigTable paper [Chang et al. 2006]. Data are stored inside structures named Columns, which in turn are contained inside Column Families, and are indexed by Key. Data is typically sharded both horizontally and vertically, by means of column families; this implies that tuples are split across several database replicas, with a much higher granularity with respect to any other kind of database.
Columnar databases are interesting in a cloud context for the following reasons:

- They are typically released in two fashions: database as a Service (DaaS) – e.g. Google Datastore, Microsoft Tables (Hybrid solutions, not completely columnar) – and standalone – e.g. Cassandra, HBase, etc.

- They are useful to store semi-structured data designed to scale to a very large size. In fact, given their partitioning strategy, they are typically used for managing Big Data that can be partitioned both horizontally and vertically.

- Thanks to projection on columns, only the data really needed are retrieved, thus maximizing the throughput.

- Some columnar NoSQL databases guarantee strong consistency under particular conditions, but, in general, they handle data in an eventually consistent way.
  Usually queries may filter on key values or column name (projection), but some of these databases permit to declare secondary indexes in order to filter queries by means of an arbitrary column value.
- Finally, some columnar share also architectural characteristics, such as the use of Memtables and SSTables, a multi-dimensional map to index properties, column families – in order to group columns of the same type – and timestamp – to maintain data up-to-date and perform versioning.

3.2 Short description of Hegira4Clouds approach

Hegira4Clouds bases the migration approach on an intermediate metamodel that defines the typical characteristics of column-based NoSQL databases. When there is a need to migrate data from a certain source to a certain target, data from the source are transformed by *Source Reading Components* (SRC), into the intermediate metamodel and queued to be sent to the *Target Writing Threads* (TWT). These read from the queue, transform the data into the format required by the target databases and store them. The metamodel is built in such a way that the approach preserves the following databases characteristic properties: *a*) secondary indexes, *b*) different data types and *c*) consistency policies. Moreover, Hegira4Clouds is able to synchronize the data between two databases, both during and after the data migration process, without any application down time. In order to do this, the API used to program the application should be able to interact with Hegira4Clouds in the event of data synchronization. For this reason, Hegira4Clouds synchronization approach works in cooperation with the CPIM library that supports the execution of CRUD (Create, Read, Update, Delete) operations on the involved databases.
In the following subsections we provide an overview of Hegira4Clouds metamodel and of its architecture.

3.3 Meta-model

By analyzing various types of columnar databases it is evident that their data model derives from the one proposed in the BigTable paper. Thus, our *intermediate Metamodel* is based on the BigTable data model and it is enriched to take into account specific aspects of some columnar databases, most notably, different consistency policies, different data types and secondary indexes. To the best of our knowledge, there are no other approaches that allow migration between NoSQL databases and that preserve strong consistency, i.e., the guarantee that all users of the database see data in the same state, as well as secondary indexes, i.e., additional way of efficiently access data, besides the typical primary keys.

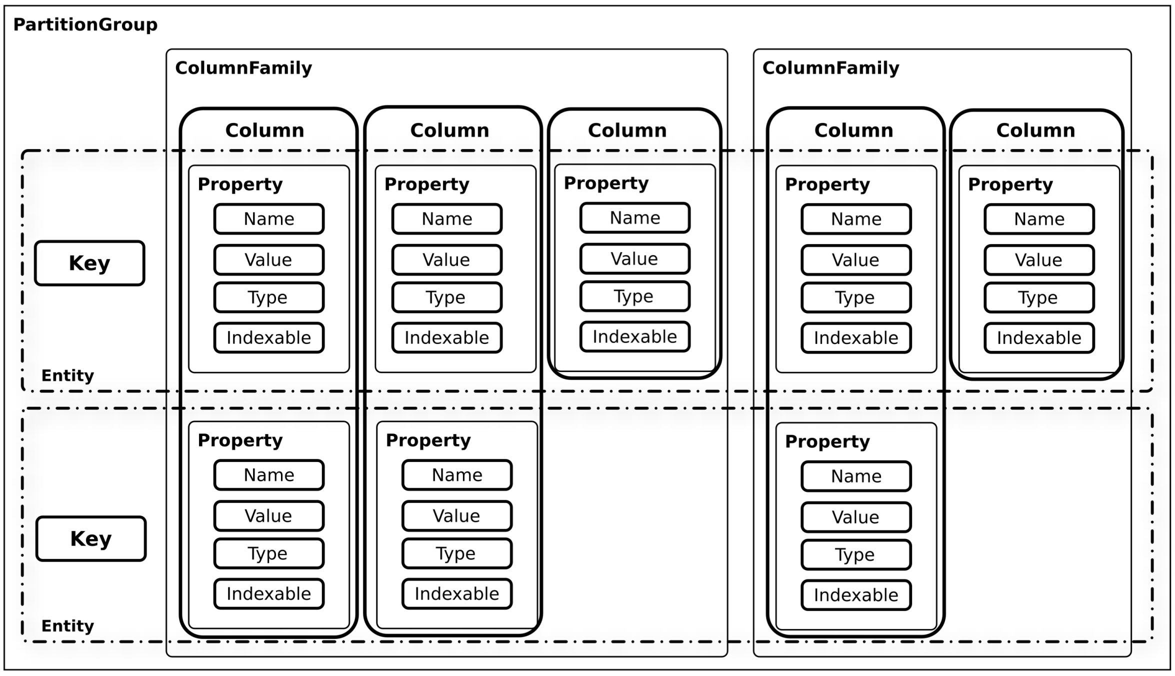

Fig. 2 – Hegira 4Clouds Metamodel

The metamodel is shown in Figure 2. *Entities* are the equivalent of tuples in a relational database and are characterized by *Entity Keys* that are used to univocally identify and index them.
Different from a tuple in a relational database, an entity can contain an arbitrary number of *properties*, which can vary from entity to entity.
Properties are the basic unit for storing data. A Property contains the single *Value* characterizing the datum to be persisted, along with its *Name*; furthermore, it provides an explicit way to declare the datum value *Type* and if that property should be *Indexable* or not.
A *Column* is a way of grouping similar properties belonging to different entities. As stated in the Google BigTable paper, *Column Families* are just groupings of different related Columns. Moreover, Column Families are disjoint, i.e. a Column may belong to just one Column Family at a time. In some databases, Column Families are used to guarantee the locality for data stored in it. Since one of the main characteristics of NoSQL databases is to store sparse data, Column Families may contain diverse number of Columns for every Entity.
Some databases support strong consistency by simply letting users model data in constructs that are specific for each database. For instance, Google Datastore uses ancestor paths, whereas Azure Tables uses a combination of Partition Key and Table Name.
For this reason, the Metamodel provides the concept of *Partition Group*. Entities inside the same Partition Group are guaranteed to be stored in the destination database, in such a way that strong

consistency (if supported) will be applicable to them on every operation performed by the destination database.

3.4 Architecture

The architecture of Hegira4Clouds is shown in Figure 3.

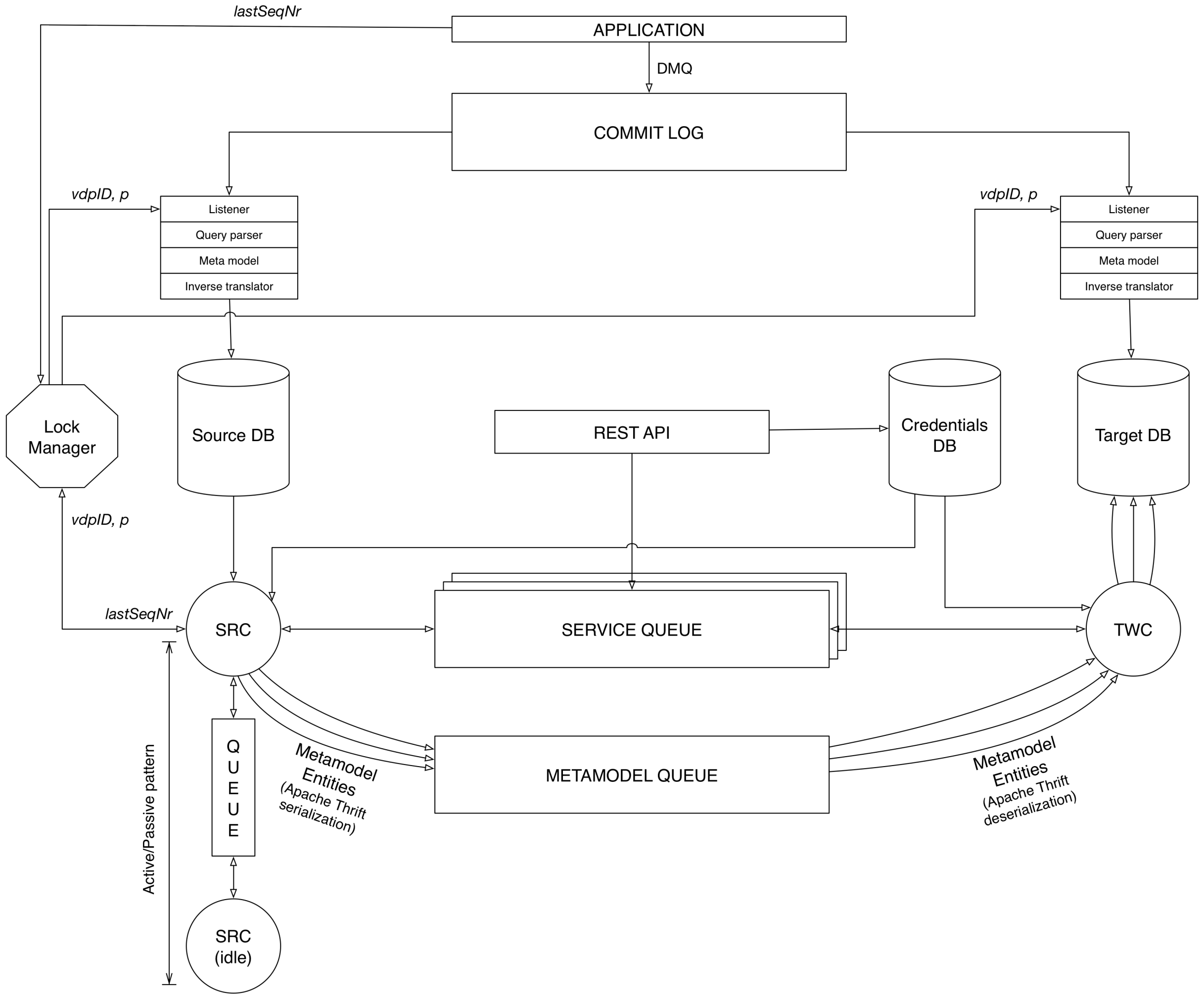

Fig. 3 – Hegira 4Clouds Architecture

From the bottom of the Figure, SRCs are the Source Reading Components that are in charge of reading data from the Source DB, translating them into the Metamodel, and storing them into the Metamodel queue. There is always a single SRC active, but, for fault tolerance reasons, a second one is kept ready to start according to the Active/Passive pattern.

On the other side of the figure we have TWC, that is, the Target Writing Component, which can contain multiple threads (the TWTs) that acquire data from the Metamodel queue, translate them and write them into the target database.

The REST API offers to the users the primitives to start and to monitor a migration and synchronization process and it exploits the Credentials DB to support authentication of users into the source and target databases to be used. The Service Queue is used to queue all commands external users can send through the REST API (e.g., migration or synchronization commands). These commands are then received by the SRC and TWC that can start their task.

Two different behaviors have to be distinguished: a) Offline data migration. b) Online data migration, with synchronization support.

Offline data migration has been discussed in details in D6.6. Here we introduce the modifications that allowed us to achieve online data migration with synchronization support.

When the migration starts, the SRC divides the dataset into chunks that are migrated independently one from the other (partitioned data migration). The chunk size is not fixed, that is, it can vary at migration time, and the user can decide it; once the chunk size is set and the migration task started, it is not possible to change the size, until a next data migration task. The list of chunks is stored in the Lock Manager together with their migration status. At the beginning a chunk is in the state "not migrated". It passes to "under migration" when the SRC starts transferring it and then to "migrated" when the TWC terminates its writing on the target database. Keeping track of this information, the Lock Manager is able to manage the synchronization of data replicas when an application performs a write/update/delete operation. More specifically, such operations are not performed anymore directly on the databases involved in the migration, but they are stored in the commit log. From this they are extracted the commit log subscribers.

The commit log subscribers are specific for each database involved in the migration and synchronization process. Once operations are extracted from the commit log they are executed according to the following approach:

- If the operation is concerning an entity that is part of a chunk that has not been migrated yet, then the operation is performed directly only on the source database.
- If the operation is concerning an entity that is part of a chunk that has been already migrated, then the operation is performed both on the source and target database.
- If the operation is concerning an entity that is part of a chunk that is under migration, the operation is blocked till the end of the chunk migration and is then performed according to point 2.

The commit log guarantees that further operations on the respective database concerning the same entity are executed in the same sequence they enter in the commit log. The approach above, together with the serial application of the operations, guarantees that the data will be consistent among the source and target databases.

Synchronization works correctly (i.e., is consistent) provided that the application using the database does not try to write directly on the database, but exploits the features offered by the Commit Log to manage write/update/delete operations and interacts with the Lock Manager to acquire a unique ID, generated in a sequential order, to be assigned to each entity.

3.5 Integration with the CPIM library

To avoid application developers having the burden of interacting with Commit Log and Lock Manager, and to guarantee that the semantics of the application, with respect to the database layer, remains unchanged, we have implemented, as part of the CPIM library, a software layer that manages this. So, the application developer only uses the JPA-based API offered by the library and does not see all issues related to the execution of the various types of queries. More details on the CPIM library can be found on D4.4.2 [Arcidiacono et al. 2015] and D4.3.3 [Abhervé et al 2015].

4 Summary of achievements

Objectives	**Achievements**
Development of a new offline data migration tool for supporting graph-based NoSQL databases.	The tool uses the basic concepts of graphs (i.e., edges and nodes) to define a common model, which is used to map data between different graph databases. The usage of a standard de-facto stack for graph based NoSQL databases (blueprints) allows the tool to support the most used graph-based NoSQL databases.
Increase fault-tolerance when migrating data across columnar NoSQL databases.	Data migration is a task that can typically last for several hours, thus it is likely that the system can incur in unpredictable faults caused, for example, by network partitions or other types of faults. As described in D6.6, the migration system was already able to tolerate faults while writing data on the target database, thus preventing data losses or inconsistency. Thanks to a preventive virtual division of the source database into chunks, the migration system is now also capable of tolerating faults while reading from the source database, thus avoiding reading the same data again, after a fault has occurred. This functionality significantly reduces both the total migration time and the costs, in the event of a fault.
Adding support for data synchronization and online data migration for columnar NoSQL databases.	Thanks to the virtual division of the source data into chunks, and the orchestration between the system components, Hegira 4Clouds is now able to provide data migration and synchronization when the cloud application is running, thus avoiding any application downtime due to the migration process. Cloud applications wanting to exploit synchronization need to send insertion and modification queries to a globally distributed commit log managed by the synchronization system. The commit log is a type of queue which guarantees the ordering of the stored messages, and, together with the synchronization system, allows to consistently and losslessly synchronize data between source and target databases. Encapsulating the interaction with the commit log, within the CPIM library (as described in D4.4.2), allows all applications built by exploiting such library to disregard completely the way data synchronization occurs. Hegira 4Cloud guarantees that the source and target databases will be eventually consistent at the end of the migration task, and that data will be kept synchronized at any point in time.

5 Conclusion

This document describes the MODAClouds tools for data migration and synchronization. The tools per se have been finalized. More work is required to i) validate their correctness and performance, ii) to integrate then within the whole MODAClouds platform in order to support the entire process of application migration from one cloud to the other, and iii) to experiment with them in one of the MODAClouds case studies.

Bibliography

[Chang et al. 2006] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: a distributed storage system for structured data,” in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI ’06. USENIX Association, 2006, pp. 15–15.

[Arcidiacono et al. 2015] Fabio Arcidiacono, Elisabetta Di Nitto, Marco Scavuzzo, Michele Ciavotta, MODACloudML package on data partition and replication - Initial version, MODAClouds deliverable D4.4.2, January 2015.

[Abhervé et al 2015] Antonin Abhervé, Marcos Almeida, Nicolas Ferry, Stepan Seycek, Elisabetta Di Nitto, MODACloudML IDE – Final Version, MODAClouds deliverable D4.3.3, March 2015.

Appendix A Data migration for NoSQL graph databases

1 Details about implementation/APIs

Data Save module, which is the tool which we are presenting within this deliverable, is designed to consume standard input in JSON format. This allows us to accommodate not only export from almost any graph database, as all of them shall provide such capabilities, but with minimal effort, allow to "graphiphy" other types of dataset, not necessarily coming from the graph databases.
Our intention was to provide a tool, which is universal enough to allow migration across different types of datasets, but structured enough to allow the user to build up nodes and its relations as predefined in the input.
Data Save module is also able to update existing graph structures if specification of the schema input is defined to do so.
As the DSS for the MODAClouds project was designed on top of ArangoDB[4] database, most of the testing was performed using this default back end. The currently supported graph database backbends are ArangoDB, Neo4j[5] and Titan[6] and each of these may require different configuration parameters.

As of the input JSON fed to the module should consist of two main sections:

1. Edges – which defines the relations between newly created or existing nodes
2. Vertices – which defines the nodes of the graph

Within the input we've introduced the "searchable" attributes, which allows the user to specify which attributes should be use in order to determine the connection or in order to decide if the input should be carried as input or update execution. Double underscore "_" sign prior to the attribute name identifies these searchable attributes.
The responsibilities of the data location and the allowance of the data migration are assumed to be out of scope of the implementation and it is carried as specified in the service level agreement or contract between the parties involved.

In case of edges, which describe the node relationship, there are 2 non-optional searchable parameters in order to ensure correctness of the schema. Those are:

- *__from* – which describe the attributes and values of the node from which the edge should be created
- *__to* – which describe the attributes and values of the node to which the edge will be connected

Furthermore, not specifying the searchable attributes on the nodes [vertices], will be treated as a input, which is the case described within the scope of this document.
Example of the input:

```
{
    "edges": [
        {
```

[4] https://www.arangodb.com
[5] http://neo4j.com
[6] http://thinkaurelius.github.io/titan/

```
            "__from": {
                "name": "A",
                "value": 1
            },
            "__to": {
                "name": "B",
                "value": 2
            },
            "label": "CONNECTED_TO"
        }
    ],
    "vertices": [
        {
            "import_id": 1,
            "name": "A",
            "value": 1
        },
        {
            "import_id": 2,
            "name": "B",
            "value": 2
        }
    ]
}
```

Vertices
Each vertex is represented by a set of key/values following the graph property model specification. Keys starting with __ "double underscore string" are meant to indicate an update action has to be carried out on that particular document. For instance:

```
{
   "import_id": 1,
   "name": "A",
   "__value": 2
}
```

This example would try an *update* on field value (setting its property value to value 2) for a vertex, which would be retrieved from a lookup by name field (if multiple vertices match this query, an error is returned and changes are rolled back), ensuring that lookup criteria match exactly one vertex e.g. query by some unique field). On the other hand, the following example would mean an *insert* action:

```
{
   "import_id": 1,
   "name": "A",
   "value": 2
}
```

All vertices must have an **import_id** property, which represents the vertex identifier. This identifier will therefore have to be stored as vertex property, since not all graph databases manage document identifiers in the same way (for instance, Neo4j does not allow to specify a vertex/edge identifier but uses an auto-increment approach).

Edges
Similar logic applies to edges objects with two concrete particularities, __from / __to and label fields. For instance, the following example would indicate an *insert* action has to be carried out:

```
{
    "property": "C",
    "__from": {
      "import_id": 1
    },
    "__to": {
      "import_id": 2
    },
    "label": "CONNECTED_TO"
}
```

__from and ***__to*** fields are set of key/values which are used to uniquely identify two vertices and create a relationship between them with a certain label. Hence, this label value must be also specified (depending on the Blueprints implementation, it can be an empty string but we encourage you to use meaningful values).
In the example above, we would query first two vertices (one with *import_id* equal to 1 and the other one with *import_id* equal to 2 and create a new edge between them with a label called "CONNECTED_TO". For updates, any field other than *__from*, *__to* and label can be marked properly (in this case, property property would be updated):

```
{
    "__property": "newC",
    "__from": {
      "name": "A"
    },
    "__to": {
      "name": "B"
    },
    "label": "CONNECTED_TO"
}
```

As required by the Description of Work, this section details the requirements and the architecture of the IDE and of its sub-components. This is an update of the requirements presented in D4.3.1 and updates in D4.3.2. For more information on the requirements to be fulfilled by the IDE, please refer to the final version of the requirements repository (D3.1.4).

2 Installation manual

There are two main methods to install and use the tool. One is to download our predefined release compiled to run on unix based systems, which includes BSD and Linux distributions. Second one, is to compile the tool from source on the desired platform:

From source

System requirements:
Please keep in mind that in order to compile this program from source, you need to have your environment ready for Scala development. Scala can be downloaded from http://www.scala-lang.org/download
Steps:

1. Clone the repository with *git clone git@github.com:CA-Labs/dss-data-save.git*

2. Navigate to the cloned repository and execute *sbt* command in the repository root folder
3. Execute *pack* task
4. Type *exit* to come back to your standard shell

Precompiled release

System requirements:
Please keep in mind that release assumes that you have Java JRE installed in your system. Java RE can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html
Steps:

- Download the latest release from https://github.com/CA-Labs/dss-data-save/releases/
- Once extracted you can execute the CLI *./bin/dss-data-save* with necessary arguments.

3 User Manual

The Data Save module expects the JSON input from stdin and requires two options:

- -d or --db: the backend used for the data graph storage. Current supported options are **arangodb**, **neo4j** and **titan**.
- -p or --properties: the absolute path to the file containing the concrete underlying graph database options in a **key/value** fashion.

As for the graph database properties, they should follow vendors specification. The distinction between the **required** and **optional** options are set as per database implementation and can be found within the documentation of the blueprints package
Here is an extract of the common pattern specification of the options:

- **Required** options will always start with blueprints.{backend}.{requiredProperty} where {backend} is the chosen underlying graph database, and {requiredProperty} is a **required** property for such database.
- **Optional** options will always start with blueprints.{backend}.conf where {backend} is the chosen underlying graph database and `{optionalProperty} is an **optional** property for such database.

The following list specifies which properties are mandatory and which ones are optional for each offered and currently supported graph database:

1. ArangoDB:

1.2. **Required** (*blueprints.arangodb.**): host, port, db, name, verticesCollection, edgesCollction

1.3. **Optional** (*blueprints.arangodb.conf.**): At this point, no further optional properties are supported.

2. Neo4j:

   1.2. Required (*blueprints.neo4j.**): directory

   1.3. Optional (*blueprints.neo4j.conf.**): see custom under (https://github.com/tinkerpop/blueprints/wiki/Neo4j-Implementation)

3. Titan
   A. **Required** (*blueprints.titan.**): storage.backend, storage.directory
   B. **Optional** (*blueprints.titan.conf.**): see custom under (https://github.com/thinkaurelius/titan/wiki/Graph-Configuration)

Once extracted you can execute the CLI *./bin/dss-data-save* with necessary arguments.

4 How to execute a migration from ArangoDB to neo4j and titan taking Venues4Clouds dataset as an example

In this example, we will present the process of migrating the ArangoDB dataset to Neo4j graph database preserving the graph structure across the data stores.

1. Dump the database to be migrated using vendors database tools.

```
arangodump --server.database ds
```

2. Prepare input data by aligning import data with import model

```
cat dump/*.data.json > import.json
sed -i -- ‘s/_key/import_id/g’ import.json
```

3. Prepare the neo4j property file

```
cat > neo4j.props << EOF
blueprints.neo4j.directory::/Users/test/data-migration
EOF
```

4. Run the migration of the data

```
cat import.json | dss-data-save -d neo4j -p neo4j.props
```

It is important to note at this stage that the shown example describes one of many ways to migrate the data. The data-save module can accommodate any of the streamed standard input which allows on the fly migrations, either application to application or application to application over network without need for intermediate file. The only requirement in this case is the need of the output to follow the established structure described in the fist chapter of this appendix.

5 How to extend the framework to support other graph databases

We have accommodated two possible approaches to the possibility of extending the module with additional databases.

- If the source graph database accommodates the blueprints framework, the only extension needed is the set of property file defined within the scope of the supported blueprints graph implementations.
- If the source database does not supporting blueprints, the only constraint applied is the availability of the JSON export tool. From our initial research, we could assure that the tools are widely available or can be developed with minimal development effort with variety of available libraries across various programing languages.

In case of custom development, the export can assure the needed model, shortening the migration steps to single execution point as the Data Save module can be embedded as a library in all JVM based languages. As for the others programing types of implementations concerning non-JVM based types of programing languages, consuming the input as **stdin** allows the tool to consume the input from almost any source. The only requirement imposed is the need of following the model of the input the data-save tool requires in order to successfully build up graph structures.

Appendix B Data migration and synchronisation for NoSQL columnar databases (Hegira 4Clouds)

1 Hegira 4Clouds APIs

As stated in previous Section, Hegira 4Cloud is made of different components each of which exposes different interfaces to the end user or to the other components.
In this section, we describe the interfaces exposed by each component.

1.1 hegira-api

This component is the entry point for Hegira 4Clouds system, it exposes a set of REST [7] interfaces for accessing a set of resources through a fixed set of operations, thus to interact with the system.
These REST APIs are available at the address: http://deib-polimi.github.io/hegira-api/.

The REST resources expose a data model that is supported by a set of client-side libraries (in different programming languages) that are made available on the files and libraries page.

There is also a WADL document describing the REST API, which is intended to simplify the reuse of the offered web services.

All endpoints act on a common set of data. The data can be represented with different media (i.e. "MIME") types, depending on the endpoint that consumes and/or produces the data. The data can be described by XML Schema, which definitively describes the XML representation of the data, but is also useful for describing the other formats of the data, such as JSON.

The page Data Model describes the data using terms based on XML Schema. Data can be grouped by namespace, with a schema document describing the *elements* and *types* of the namespace. Generally speaking, *types* define the structure of the data and *elements* are *instances* of a type. For example, *elements* are usually produced by (or consumed by) a REST endpoint, and the structure of each element is described by its *type*.

1.2 hegira-components

This component is the core of Hegira 4Clouds migration system. In fact, it allows to launch both the Source Reading Component (SRC) and the Target Writing Component (TWC).
It is possible to interact with it by means of the command line interface (CLI).
The interfaces exposed by the component are described in its repository README file, available at the address: https://github.com/deib-polimi/hegira-components.

1.3 hegira-CLsubscriber

This component is the core of Hegira 4Clouds synchronization system. In fact, it allows to launch the commit-log subscriber both for the source and target databases.
It is possible to interact with it by means of the command line interface (CLI).

[7] http://en.wikipedia.org/wiki/Representational_state_transfer

The interfaces exposed by the component are described in its repository README file, available at the address: https://github.com/deib-polimi/hegira-CLsubscriber.

2 Installation manual

In this section we show how each of the components, described in previous Section, should be installed. In order to successfully install and deploy the components, we assume the following software have already been installed, configured and, if necessary, started:

- JDK 1.7 or greater, on all the machines that are going to be used to install Hegira 4Clouds components.
- Apache Tomcat 7 or greater, only on the machine hosting hegira-api component.
- Apache Maven 3.2.2 or greater, on the machine that will compile the source code into binaries.
- RabbitMQ 3.3.6 or greater.
- Apache ZooKeeper 3.4.6 or greater.
- Apache Kafka 0.8.2 or greater, together with Hadoop 2.10

Disclaimer: fault-tolerance guarantees described in previous deliverables depend also on the fact that RabbitMQ, Apache Kafka and Apache ZooKeeper have been properly configured and deployed. The minimum deployment required consists in at least three replicas, per each component. The specific details to set each component replication factor can be found in the specific component documentation[8].

2.1 hegira-api installation and configuration

Hegira-api component can be downloaded from the Git repository at the following address:
https://github.com/deib-polimi/hegira-api/.
The README file in the repository contains the information necessary to build the application.
Once built, an Application Server, e.g. Tomcat, should execute the component.
The instructions to deploy the component on Apache Tomcat are available in the README on the repository.

2.2 hegira-components installation and configuration

Hegira-components can be downloaded from the Git repository at the following address:
https://github.com/deib-polimi/hegira-components.
The README file in the repository contains the information necessary to build and execute the application.

2.3 hegira-CLsubscriber installation and configuration

hegira-Clsubscriber can be downloaded from the Git repository at the following address:
https://github.com/deib-polimi/hegira-CLsubscriber.
The README file in the repository contains the information necessary to build and execute the application.

[8] http://kafka.apache.org/documentation.html#brokerconfigs
http://www.rabbitmq.com/distributed.html
http://zookeeper.apache.org/doc/r3.4.6/zookeeperAdmin.html#sc_zkMulitServerSetup

3 User manual

3.1 How to run the system

As previously seen, Hegira 4Clouds is mainly composed of 3 components, one (hegira-api) which exposes a set of interfaces to interact with the whole system, another one (hegira-components) provides the data migration capabilities and one (hegira-CLsubscriber) which allows the two databases to stay in synch during and after a data-migration task.

Assumption 1: in order to perform data synchronization tasks the application utilizing the source database must have persisted data on it only by means of the CPIM library (described in Deliverable D4.4.2). The reason for this restriction, as explained in Section 3.5, is to simplify the work of designers who do not have to take care of the management of Hegira4Clouds Commit Log during synchronization.
Assumption 2: in order to correctly retrieve data after a data migration task, since the target database may not support all of the data-types used by the source database, it is advisable to use the CPIM library because the library guarantees that all data can be read back even if the original data-type is not supported by the target database.

Each component can be started independently from the other, in the same or in different machines. However, empirical tests have suggested that the optimal configuration setup consists in placing, when possible, the migration components in the same data-centre of the target database. In case this is not possible, the migration might be slow down because of larger network latency.

System deployment

1. Rest API deployment: the instruction to deploy hegira-api component are reported at https://github.com/deib-polimi/hegira-api#deploy
2. Migration system deployment: the instructions to run the migration components are reported at https://github.com/deib-polimi/hegira-components#usage.
   In particular, the executable should be run two times, once for the source and once for the target database.
3. Synchronization system deployment: the instructions to run the synchronization components are reported at https://github.com/deib-polimi/hegira-CLsubscriber#usage.

In particular, the executable should be run two times, once for the source and once for the target database.

System start-up

The system can be instructed to perform just a data migration task (without any further synchronization) or it can start performing data migration and synchronize all further writes on both databases. In order to do so, hegira-api component exposes two different resources (switchOver and switchOverPartitioned), which are documented at: http://deib-polimi.github.io/hegira-api/resource_API.html.

Once one of the two resources has been called, Hegira 4Clouds starts to migrate data.

Disclaimer: In case only data migration has been chosen, none of the applications should write data on the source and target databases during the migration task; doing otherwise would affect data consistency. Instead, in case also data synchronization has been chosen, applications can continue to read and write data **only by means of the CPIM library**; in this case, during the data migration task, the application should take into account a possibly higher consistency delay, with respect to the one the source database provided, due to the synchronization system. That is, the applications should expect to read stale data, in some limited cases, for a slightly higher period of time with respect to previous read operations (when the migration and synchronization tasks were not started yet). The two databases will eventually be consistent at the end of the migration task.

4 How to execute data migration and synchronization from GAE Datastore to Azure Tables

In this Section we show two examples of usage:

1. Simple, offline data-migration between Google AppEngine Datastore and Azure Tables.
2. Online data-migration and synchronization between Google AppEngine Datastore and Azure Tables.

In both examples, we suppose the following:

- The component hegira-api has been deployed on a Tomcat application server, reachable at the following address: http://10.0.0.1:8080
- A RabbitMQ broker is reachable through the following address 10.0.0.2
- An instance of Apache ZooKeeper is listening on the following address 10.0.0.3:2181
- An Apache Kafka broker is reachable at 10.0.0.4:9092
- All Hegira 4Clouds components have been compiled and configured with the above information and the proper credentials to access the databases.

Offline data-migration example

```
java -jar hegira-components -t SRC -q 10.0.0.2
```

We begin by starting hegira-components under SRC mode:

Then, we execute hegira-components under TWC mode:

```
java -jar hegira-components -t TWC -q 10.0.0.2
```

At the end, we issue the following POST request to hegira-api component:

```
http://10.0.0.1:8080/hegira-
api/switchover?source=DATASTORE&target=TABLES&threads=32
```

The above request tells Hegira 4Clouds to perform offline data migration from GAE Datastore (source=DATASTORE) to Azure Tables (target=TABLES), using 32 threads that write in parallel on

Azure Tables. Credentials, as well as databases endpoints, for accessing the respective database are stored in the credentials file contained by hegira-components.
hegira-api component will respond with a Status object which acknowledges that the offline migration has begun.

Online data-migration and synchronization example

The hegira-components can be started in the same way shown in the previous example.

Then, the hegira-CLsubscriber can be started, at first in *src mode* and after in *dst mode,* passing the source and target database, respectively, as well as ZooKeeper address and port:

```
java -jar hegira-CLsubscriber -m src -d DATASTORE -z
10.0.0.3:2181
```

```
java -jar hegira-CLsubscriber -m dst -d TABLES -z 10.0.0.3:2181
```

At the end, we issue the following POST request to hegira-api component:

```
http://10.0.0.1:8080/hegira-
api/switchoverPartitioned?source=DATASTORE&target=TABLES&threa
ds=32&vdpSize=2
```

The above request tells Hegira 4Clouds to migrate and synchronize data from GAE Datastore to Azure Tables, using 32 threads that write in parallel on Azure Tables; the migration is performed in a partitioned way (see the explanation in Section 3.4), with a chunk size equal to 10^2 entities.

hegira-api component will respond with a Status object which acknowledges that the online migration has begun.

5 How to extend the framework to support other columnar databases

In order to extend Hegira 4Clouds to support a new database, several changes need to be done on each component.

In particular, since hegira-api component should be aware of all the supported databases, it needs to be modified as described in the wiki: https://github.com/deib-polimi/hegira-api/wiki/Extension.

The main extension point is the application hegira-components, here the developer should add an adapter to properly communicate with the new database, together with two translators, one, which converts the data from the new database data-model to the intermediate meta-model, and another which converts the data from the intermediate meta-model towards the new database data-model (more details on this are provided on D.6.6). This extension is covered more in details in the wiki: https://github.com/deib-polimi/hegira-components/wiki/Extension. Moreover, the relative JavaDoc can be found at the following address: http://deib-polimi.github.io/hegira-components/.

Extending the hegira-CLsubscriber component to support a new database consists in copying the adapter and the transformers, contained in the hegira-components application and performing the changes described in the wiki page: https://github.com/deib-polimi/hegira-CLsubscriber/wiki/Extension.