Experiences and challenges in building a data intensive system for data migration

Marco Scavuzzo[1] · **Elisabetta Di Nitto**[1] ·

Abstract *Data Intensive (DI) applications* are becoming more and more important in several fields of science, economy, and even in our normal life. Unfortunately, even if some technological frameworks are available for their development, we still lack solid software engineering approaches to support their development and, in particular, to ensure that they offer the required properties in terms of availability, throughput, data loss, etc.. In this paper we report our action research experience in developing-testing-reengineering a specific DI application, Hegira4Cloud, that migrates data between widely used NoSQL databases. We highlight the issues we have faced during our experience and we show how cumbersome, expensive and time-consuming the developing-testing-reengineering approach can be in this specific case. Also, we analyse the state of the art in the light of our experience and identify weaknesses and open challenges that could generate new research in the areas of software design and verification.

Keywords Data intensive applications · Experiment-driven action research · Big data · Data migration

Communicated by: Luciano Baresi, Tim Menzies and Andreas Metzger

✉ Marco Scavuzzo
marco.scavuzzo@polimi.it

Elisabetta Di Nitto
elisabetta.dinitto@polimi.it

Danilo Ardagna
danilo.ardagna@polimi.it

[1] Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy

This is a post-peer-review, pre-copyedit version of an article published in Empir Software Eng (2018) 23:52–86. The final authenticated version is available online at: https://doi.org/10.1007/s10664-017-9503-7

1 Introduction

In general, many sectors of our economy are more and more guided by data-driven decision processes. A recent analysis (Marr 2015) has shown how Big Data could produce $300 billion potential annual value to US health care, while Europe public sector could potentially reduce expenditure of administrative activities by 15–20 %, with an increase of value ranging between $223 to $446 billion (Manyika et al. 2012; Chen and Zhang 2014).

Of course, Big Data are effective if they are created, analyzed, and transformed. Systems in charge of these tasks are typically called *Data Intensive* (DI).

From the software engineering perspective, DI applications must tolerate hardware, software and network failures, they should scale with the amount of data, and be able to elastically leverage additional resources as and when they are provided (Kambatla et al. 2014). Moreover, they should be able to avoid data drops introduced in case of sudden overloads and should offer some Quality of Service (QoS) guarantees.

Ensuring all these properties is, in general, not an easy task. In fact, the practice is not well established and, thus, identifying the right solution can require several rounds of experiments and the adoption of many different technologies. This implies the need for highly skilled persons and the execution of experiments with large data sets and a large number of resources. Therefore, it requires a significant amount of time and budget.

In this paper we aim at highlighting the issues we faced in the development of a relatively simple, but significant DI application called Hegira4Cloud.[1] It is a system supporting the offline migration of large data sets stored in NoSQL databases, including so-called *Database as a Service* (DaaS) like Azure Tables, Google Datastore and on-premise NoSQL databases like Apache Cassandra and Apache HBase. Hegira4Cloud is representative of the category of DI systems because it has to handle large volumes of data with different structures and has to guarantee that their important characteristics, in terms of data types and transactional properties, are preserved. Also, it poses stringent requirements in terms of correctness, high performance, fault tolerance, and fast and effective recovery.

Looking at the state of the art in the area of software engineering for DI applications (as we discuss in Section 2.1), we could not find an approach that could fulfill our needs, so we had to adopt a design and development approach based on action research (Baskerville and Myers 2004; O'Brien 1998). During this experience, we encountered several problems that we could observe and address only by adopting a time-consuming development-testing-reengineering approach, for which we could not rely on the support of high-level design, analysis and testing tools.

Hegira4Cloud is the result of a two years work. We have published partial results in Scavuzzo et al. (2014, 2016). In particular, in Scavuzzo et al. (2014) we have presented the metamodel that we adopt to handle the differences between the different data models supported by NoSQLs and we have described the first version of Hegira4Cloud. In Scavuzzo et al. (2016) we have given a brief and intuitive overview of the improvements introduced in Hegira4Cloud to increase performance and fault-tolerance. The original contribution of this paper is a thorough description of the trial-and-error, action-research approach that allowed us to make Hegira4Cloud scalable (in terms of volume of data that can be migrated),

[1] Repositories:
Monolithic version: https://github.com/deib-polimi/Hegira4Cloud
Improved prototype: https://github.com/deib-polimi/hegira-components
Rest API: https://github.com/deib-polimi/hegira-api

performant and resilient to internal and external faults. In order to do so, we extensively present and discuss all the problems encountered during the process, which led to some important decisions during the software design and engineering process and, which can be useful for other practitioners as well.

The rest of the paper is organised as follows. Section 2 presents the state of the art. Section 3 provides a general overview of Hegira4Cloud. Section 4 presents the first version of the tool and the corresponding experimental results. Section 5 describes the improvements that have led to the second version of Hegira4Cloud and the experimental results that demonstrate the correctness and performance of the migration approach, but highlight some issues concerning fault tolerance. Moreover, Section 6 presents the way we have addressed these last concerns in the third version of the system. In Section 7, based on our experience with the multiple development-testing-reengineering iterations around Hegira4Cloud, we summarize the challenges we faced, we discuss the limitations of available tools and identify new research areas for future research. Finally, in Section 8 we draw the conclusions.

2 State of the Art

In the first section of this state of the art analysis we discuss about the endeavours in the software engineering arena that aim at supporting the design of DI applications. In the second and third section we focus on the literature that is specifically related to Hegira4Cloud. In particular, in Section 2.2 we introduce the main characteristics of NoSQL databases and their relevance in the context of DI systems, while in Section 2.3 we describe the state of the art concerning data migration.

2.1 Engineering Data Intensive Applications

From the point of view of technology, during the last decade, several Big Data processing frameworks have been developed and, in many cases, released as open-source projects. These frameworks ease the development of data intensive applications. They are typically classified in *batch processing* and *stream processing* frameworks, according to the type of workload they need to cope with. Some famous examples include Hadoop,[2] Spark[3] and Flink.[4] These frameworks greatly simplify the development of data intensive applications, in that they provide relatively easy to use programming models and interfaces to define applications; the developer does not need to worry about tasks parallelization and scheduling aspects, since they are typically handled by the framework itself. Notwithstanding, several complexities, that arise when designing specific data intensive applications, are not directly supported by these frameworks. These complexities include, for example, *i)* the dimensioning of the computational resources to allot to the framework; *ii)* the interfacing to the storage systems typically providing non-standard interfaces and data models; *iii)* the variety (schema-less data extracted from different sources) of data to be processed; *iv)* the lack of clear storage systems SLAs that inevitably results in trial-and-error procedures to roughly derive them.

[2] http://hadoop.apache.org/

[3] http://spark.apache.org/

[4] http://flink.apache.org/

All the above issues require for proper guidelines and supporting tools to allow designers to identify proper solutions that combine different frameworks and technologies. An interesting initiative to define an architectural framework for DI applications has been proposed by the NIST Big Data Working Group and is called NBDRA (NIST Big Data Reference Architecture) NIST Big Data Interoperability Framework (2015). NBDRA defines the conceptual components that belong to a DI applications (data collection, preparation/curation, analytics, visualization, access) and highlights the infrastructural/technological conceptual elements that are relevant to its operation. It is interesting as it is general enough to capture the main characteristics of most DI applications, including Hegira4Cloud, and positions the various available technologies and frameworks in a coherent framework. As such, it constitutes a very good guide to the identification of the main functional components of DI applications and to the selection of the classes of technologies needed for its implementation. However, it cannot help in the specific design activities devoted to fine tuning and performance optimization.

Szyperski et al. (2016) highlight that constructing large big data solutions is still an engineering challenge that requires good ways of managing oceans of resources and of addressing contradictory requirements concerning collocation, consistency and distribution.

Gorton and Klein (2015) highlight the utility of adopting architectural tactics, that is, design options that embody architectural knowledge on how to satisfy specific concerns and propose some initial tactics for big data applications that address performance, scalability and availability. While certainly the idea of tactics appears to be quite useful and promising, the specific ones that are proposed in the paper seem to be specific of a web-based architecture composed of web or application servers on one side and of a replicated and partitioned database on the other.

In Szyperski et al. (2016) Barga argues that in a big data context good design is needed to achieve efficient resource management and, thus, scalability and suggests the adoption of a number of design principles such as the opportunity to privilege asynchronous communication vs. synchronous, limit the usage of contentious resources, identify partitions for resources. In our work we did adopt such design principles, but, as we discuss in the rest of the paper, we realized that their application is not trivial and still requires significant experimentation and fine tuning.

2.2 NoSQL Databases

The NoSQL movement was born around the concept that a single type of database (i.e., RDBMSs) is not the best solution for all the types of applications (Stonebraker and Cetintemel 2005). In particular, in order to access data quickly, the database should be chosen based on the problem the application is aiming to solve. In fact, RDBMSs are based on the Entity-Relation model and SQL is the standard query language for them. They allow a database designer to minimize data duplication (redundancy) within a RDBMS through the process of normalization (Kent 1983). Also, the Entity-Relation model forces these databases to use a strict schema, which limits data evolution and makes it difficult to partition data. In order to overcome these limitations, NoSQL databases (Sadalage and Fowler 2012) typically provide none or flexible schema management capabilities and make the assumption that data are stored in a denormalized way. Denormalization reduces query execution time at the cost of potential data duplication (data is duplicated instead of referenced) and allows easier partitioning.

Moreover, NoSQL databases try to handle sever other business problems that traditional RDBMSs are not able to efficiently solve (Stonebraker and Cetintemel 2005; Stonebraker et al. 2007; Harizopoulos et al. 2008), namely:

- **Data volume**: currently, enterprises are gathering Terabytes of data that need to be processed and RDBMS storage models are not efficient in storing or retrieving huge amount of data. Usually, queries against relational tables require accessing multiple indexes, joining and sorting rows from multiple tables.
- **Data variety**: data usually come from diverse sources and in diverse forms, i.e., structured, semi-structured and unstructured.
- **Data velocity**: data need to be written and accessed as fast as possible in order to provide either fast transaction processing capabilities (OLTP) or fast data analytic capabilities (OLAP).

Managing huge quantities of data typically requires to split the data onto separate database *nodes*; NoSQL databases automatically manage and distribute the data to such nodes, transparently from the user's perspective. Data sharding (or horizontal partitioning) inherently improves the database availability, since users may access only the nodes that contain the data they are interested into. To further improve availability and fault-tolerance, NoSQL database vendors typically replicate each node (*replica*). Thus, NoSQL systems are categorized also according to their replication scheme: data are replicated within the same data-center or geographically across multiple data-centers (Armbrust et al. 2010).

Generally NoSQL databases do not provide transactional properties in the common sense of the word (i.e., like in RDBMSs). In fact, the concept of transactions in NoSQL databases is typically limited to a single entity (or tuple), meaning that only group of operations (reads or writes) on a single entity can benefit of some guarantees similar to RDBMSs ACID properties. Such guarantees can, in fact, be a combination of the *atomicity*, *isolation* and *consistency* properties and they are actually dependent on the specific NoSQL database being used.

In terms of *consistency*, NoSQLs (in order to provide high-availability and cope with the complexity of distributed resource management) typically trade strict consistency models (e.g., serializability) in favour of weaker types. The most used form of consistency provided by current NoSQL databases is *eventual consistency*. According to this model, all updates, executed on a node, will eventually be propagated to its replicas. Hence, there is no guarantee that, at any point in time, multiple read operations on the same tuples will return consistent values. More recently, due to the increasing need of several applications to achieve more flexible consistency guarantees, some NoSQL implementations have been offering other types of consistency policies each of which depends on the specific database design choices and can be configured either at *insertion-time* (i.e., at the data model level) or at *query-time* (with different granularity, ranging from query-specific consistency – e.g., in MongoDB – to connection-specific consistency – e.g., in Apache Cassandra). One of these policies is *strong consistency*, which implies that parallel read operations can observe, at any point in time, only one consistent state. In Abadi (2012), authors explain in detail the effects of operation replication strategies in distributed databases, thus highlighting the trade-offs between latency and consistency.

In the event of a network partition (causing an arbitrary loss of messages), distributed NoSQL databases must trade availability against consistency, according to Eric Brewer's CAP theorem (Brewer 2012); this has favoured the development of hundreds of database, some of them are CP (preferred consistency over availability), some others are AP (preferred availability over consistency).

In terms of *atomicity*, NoSQLs guarantee that group of operations on a single entity will either be applied as a unit to the entity or, if any operation fails, none of them will be applied.

Finally, in terms of *isolation*, NoSQLs guarantee that each operation within the same group on a single entity will be executed serially (i.e., operations concurrency control is applied on an entity basis).

Google Datastore[5] provides also a transactional behaviour that is not limited to a single entity as in most NoSQLs, but can span on multiple entities at the same time. This behaviour can be achieved through the concept of *ancestor path*. An ancestor path represents a hierarchical relationship among entities, i.e., upon entity creation on the Datastore, the user can optionally create a relationship between this entity and another one (a parent), which, recursively, may have another parent. For example, imagine to have three entities representing three different people, e.g., Bob, Alice and John; Bob is Alice's son and Alice is John's daughter. To represent this information in Google Datastore the user must declare Alice entity as Bob's parent entity and John entity as Alice's parent entity; by doing so the user creates an ancestor path relationship among the three entities, which goes from Bob to Alice and to John. In this case, every operation, expressed on entities belonging to the same ancestor path, will benefit of the atomicity, isolation and strong consistency properties. This optional behaviour, of course, limits the write throughput because of the synchronous replication of the operations over the replicas containing the data.

NoSQL databases also show a large variety of data models. Even though there is no generally accepted classification, the most accredited one (Sadalage and Fowler 2012) distinguishes between *Key-Value*, *Document*, *Graph*, and *Column-family-based* NoSQL databases. For the purpose of this paper it is interesting to mention that Key/Value stores are similar to data structures like Maps or Hash-tables, where values are typically stored in an uninterpreted, compressed form and queries are performed only by key.

Moreover, column-family-based databases can be seen as multidimensional maps. These databases owe their name to the data model proposed by the Google BigTable paper (Chang et al. 2006). According to this model, data are stored in columnar format (in contrast with typical row-based format) inside structures named *Columns*, which, in turn, are grouped inside *Column Families*, and are indexed by *Keys*. Data can be semi-structured, which means that not all columns need to be filled in for each tuple and that a column can contain data of different types. Column-family-based NoSQLs are particularly interesting for their ability to support high performance data analytics tasks (OLAP) and to enable scalability at the growth of data volume by partitioning data by column families. Some of these NoSQL databases guarantee strong consistency under particular conditions (e.g., Google Datastore as mentioned above), but, in general, they handle data in an eventual consistent way. Usually queries filter data on key values or column name (projection), but some of these databases also permit to declare secondary indexes, in order to build queries that can efficiently filter data by means of an arbitrary column value.

[5] https://cloud.google.com/datastore/

In a cloud computing context NoSQLs are typically released in two fashions, as Database as a Service (DaaS) – e.g., Google Datastore and Microsoft Azure Tables[6] – and standalone – e.g., Apache Cassandra,[7] Apache HBase,[8] etc..

2.3 Data Migration

We define *data migration* as *the task of creating and transferring a snapshot of a source database into the target database(s), adapting it according to targeted technologies characteristics, thereby making that snapshot portable across different representatives of the NoSQL landscape* (Scavuzzo et al. 2016). Additionally, if no manipulation operations (i.e., inserts or updates) are allowed on the databases, while a data migration task is in process, we say that the data migration is performed offline.

Data migration is a well established topic for relational databases (see, e.g., Oracle SQL developer,[9] BoxFuse Flyway,[10] Liquibase,[11] MySQL Workbench: Database Migration,[12] IBM InfoSphere Data Replication and Federation,[13] (Ceri and Widom 1993; Atzeni et al. 2012)), thanks to *i)* the standardization occurred at the data model level, *ii)* the presence of a data schema (expressed by means of a DDL) and *iii)* a (almost) standard data access and manipulation language (i.e., SQL).

In the NoSQL database field, data migration is complicated by the lack of a homogeneous data model and data schema, and, to the best of our knowledge, there is no tool which supports complete (2-ways) data migration across different NoSQLs in a general way. Some databases provide tools (e.g., Google Bulkloader[14]) to extract data from themselves or to import data exclusively from formatted text files (e.g., CSV, JSON), but in the end, it is up to the programmer to actually map those data to the target database data model and perform the migration. Other tools Neo4J import tool,[15] MongoDB mongoimport tool[16] just import data from formatted text files and, as a consequence, they rely on other third-party applications to extract data from the source database and to store them in such text formats. Some databases provide tools (e.g., DocumentDB Data Migration Tool[17]) capable of importing data from several sources (formatted text files or other databases) and exporting them into text files, but these tools do not support automatic mapping of incompatible data types across different databases and do not provide any fault-tolerance mechanism; additionally, for some of them

[6] https://azure.microsoft.com/en-us/services/storage/tables/

[7] http://cassandra.apache.org/

[8] http://hbase.apache.org/

[9] http://www.oracle.com/technetwork/database/migration/index-084442.html

[10] https://github.com/flyway/flyway

[11] http://www.liquibase.org

[12] http://www.mysql.it/products/workbench/migrate/

[13] https://www-01.ibm.com/marketing/iwm/iwm/web/signup.do?source=ibm-analytics
&S_PKG=ov4921&S_TACT=M161001W&dynform=9816

[14] https://chromium.googlesource.com/external/googleappengine/python/+/200fcb767bdc358a3acb5cf7cad1
376fe69f12c5/google/appengine/tools/bulkloader.py

[15] http://neo4j.com/docs/stable/import-tool.html

[16] https://docs.mongodb.org/manual/reference/program/mongoimport/

[17] http://goo.gl/Z307aS

(e.g., MongoDB mongoimport and mongoexport tool,[18,19]) the use in production is explicitly discouraged.

Another type of technology recently emerged for data migration and replication is that of data change and capture systems. These systems can typically replicate data across proprietary products, as well as SQL databases (e.g., Oracle Streams Wong et al. (2009)), by relying on the possibility to access databases redo-logs. To the best of our knowledge, the only tool that also supports heterogeneous NoSQL databases is Databus (Das et al. 2012). However, Databus only supports data sources that provide a time-line consistent change stream that can be tapped from. Therefore, for example, any other Database as a Service or proprietary database cannot be supported by Databus (or by other similar systems), unless they expose their change stream to it (i.e., until the DaaS vendor decides to support it). Additionally, since Databus can only replicate data contained in the change stream, all the data stored in any data source before Databus was actually started and that is not available in the change stream, cannot be replicated.

3 Hegira4Cloud Overview

3.1 Main Characteristics and Requirements

Hegira4Cloud is a batch processing DI application focusing on offline data migration between column-family-based NoSQL databases, including two DaaS solutions, i.e., Google Datastore and Azure Table. It addresses specifically high volume, velocity and variety of data. More in detail, the volume of data we handle is in the order of hundreds of Giga bytes; the velocity of operation – i.e., data migration in our case – should be as high as possible to cope with the continuity of service requirements of the end user applications that exploit the data subject to migration. The variety of data is not controlled by Hegira4Cloud itself as it depends on the data to be migrated, but it has to be preserved by Hegira4Cloud during the data migration.

Hegira4Cloud implementation must fulfill the following main requirements:

- The migration process is correct, i.e., all data are transferred from the source to the target NoSQL and, if data are transferred back to the source NoSQL, they are fully equivalent to the original ones.
- The migration terminates in a reasonable amount of time. By “reasonable amount of time” we mean that, given a source and a target database, the time needed to migrate all the data between the two is at most twice of the maximum value between the time that it would take to read all the data from the source database and the time that it would take to write the same amount of data to the target database.
- If a fault occurs, the platform is able to recover and to continue the migration process without data loss.
- It is easy to incorporate into the platform a new database from/to which data should be migrated.

[18] https://docs.mongodb.com/manual/reference/program/mongoimport/

[19] https://docs.mongodb.com/manual/reference/program/mongoexport/

These four requirements are quite challenging, especially into a context in which: (i) the amount of data to be transferred can be in the order of some Giga or Tera Bytes; (ii) the resources to run a NoSQL or the NoSQL DaaS itself are paid per use, and thus, in case of failure, multiple attempts to repeat a migration could have a significant cost; (iii) NoSQLs are continuously evolving and are quite different one from the other.

3.2 Hegira4Cloud Meta-model

Hegira4Cloud relies on an intermediate format for representing data under migration that is defined by the *intermediate meta-model* shown in Fig. 1. It follows quite closely the structure of Google BigTable and takes into account the data models and features of the most widely used NoSQLs. Entities in our meta-model are similar to the concept of tuples in RDBMSs, but while tuples in the same table have all the same structure, entities can contain an arbitrary number of properties that are grouped in the same column when they refer to the same concept. Each property is characterized by a *Name*, *Value*, *Type* and can be *Indexable* or not. Entities can be sparse, in the sense that they can miss some properties in some columns (see the case of the second entity in Fig. 1). Also, in the same column, it is possible to have properties with different names and types. Additionally, columns can be organized in *Column Families* that, in turn, can belong to the same or different *Partition Groups*. Finally, each entity includes a unique *Key*.

Thanks to this intermediate meta-model, Hegira4Cloud is able to preserve the data types, consistency policies, and secondary indexes supported by the source NoSQL even in the case the target database does not offer these features. More in detail:

- *Data type preservation* is accomplished by keeping track explicitly of the type of migrated data, even though that type is not available in the destination NoSQL. To do so, data converted into the intermediate format are always serialized into the property Value field and the original (fully qualified) data type is stored as a string into the property Type field. When data are converted from the intermediate format into the target

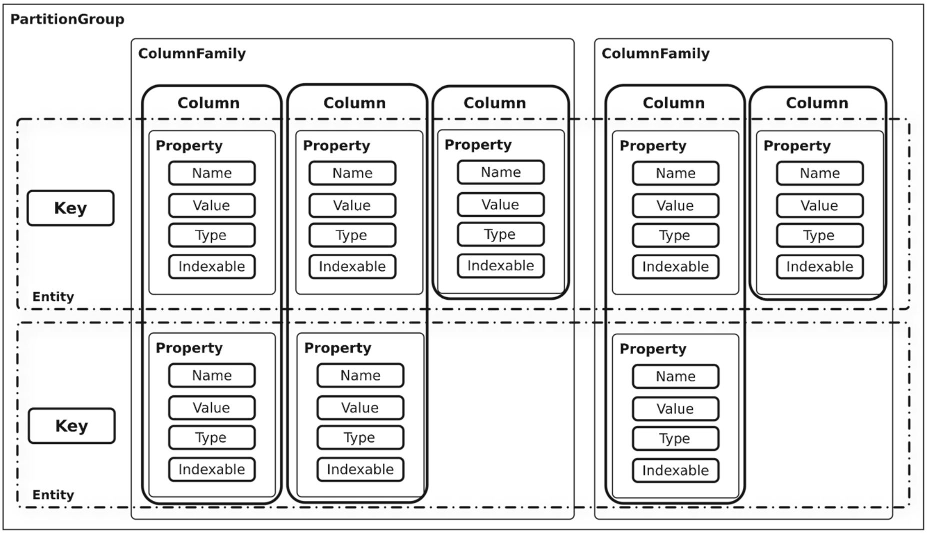


Fig. 1 Intermediate metamodel

one, if the destination NoSQL supports that particular data type, the value is deserialized. Otherwise, the value is kept serialized and it is up to the user applications to correctly interpret (deserialize) the value according to the corresponding Type field that is kept stored separately as metadata information.

- *Consistency policies* are handled through the concept of *Partition Group*. Entities that require strong consistency will be assigned, in the intermediate format, to the same Partition Group. Entities managed according to an eventual consistency policy will be assigned to different Partition Groups. More in detail, when converting into the metamodel format entities that are related with each other, Hegira4Cloud assigns them to the same Partition Group. When entities share the same Partition Group, if the target database supports strong consistent operations, then Hegira4Cloud adapts data accordingly, depending on the target database data-model. For istance, if the target database is Google Datastore, all data in the same Partition Group are organized in the same ancestor path hierarchy. Otherwise, Hegira4Cloud simply persists the data so that they will be read according to the target database default consistency policy, and creates an auxiliary data structure to preserve the consistency information, so that strong consistency can still be applied if data are migrated again into a database that supports strong consistency (Scavuzzo et al. 2014).
- *Secondary indexes* are preserved across different NoSQLs by means of the property Indexable field. More specifically, during the conversion into the intermediate format, if a certain property needs to be indexed, it is marked as indexable. When converting it into the target format, if the target NoSQL supports secondary indexes, the property is associated to an index according to the specific interfaces provided by the target database. Otherwise, Hegira4Cloud creates an auxiliary data structure on the target NoSQL, which stores the references to the indexed properties, so that, when migrating again these data to another database supporting secondary indexes, they can be properly reconstructed.

A more detailed presentation of the intermediate meta-model can be found in Scavuzzo et al. (2014), where we have shown that such meta-model is sufficiently general for dealing with the features of so-called column-family based and key-value NoSQL databases (Scoffield 2010; Popescu 2010).

An alternative approach to the storage of the meta-data, generated from the migration system, into the target database is currently under consideration. In fact, when the target database does not support some of the properties expressed in the meta-model representation, and originally present in the source database, such new approach would store the above mentioned auxiliary meta-data within the migration system itself, instead that in the target database. This approach would require the applications using the target database to interact with the migration system in order to retrieve the meta-data and properly operate on the target database.

4 Hegira4Cloud v1

4.1 High Level Architecture

The very first Hegira4Cloud monolithic architecture is shown in Fig. 2. It is implemented in Java because this language is the most widely supported by NoSQL access libraries. The *Migration System Core*, contains two processes: a *Source Reading Component (SRC)* and

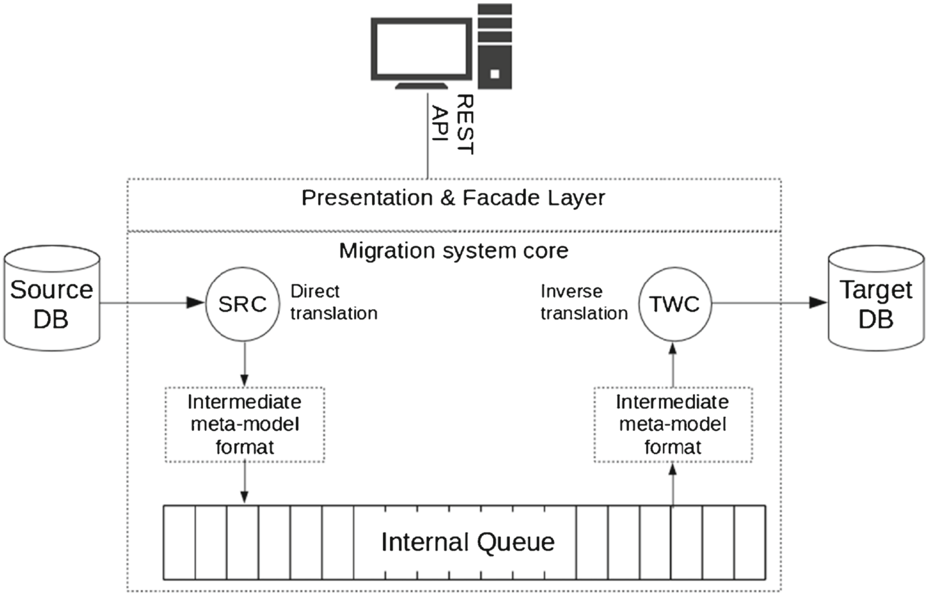


Fig. 2 Hegira4Cloud v1 architecture

a *Target Writing Component (TWC)*. The SRC extracts data from the source database, one entity at a time or in batch (if the source database supports batch read operations), performs the conversion into the intermediate format, by means of a *direct translator*, and puts the data in an internal queue. The queue is used as a buffer, in order to decouple the reading and writing processes, thus managing the different throughput of the source and target NoSQL. When data are in the queue, the TWC extracts and converts them into the target NoSQL data model, thanks to an *inverse translator*, and stores these data into the target NoSQL.

Direct and inverse translators are specific for each database supported by Hegira4Cloud and can be easily included in the platform as plugins. In order to support migration to/from a new NoSQL, the corresponding direct and inverse translators have to be developed. Thanks to the intermediation of the meta-model, the number of translators grows linearly with the number of supported databases, in contrast to a direct mapping approach between different databases models that would let the number of translators grow quadratically (Cluet et al. 1998).

4.2 Preliminary Experiments

In Scavuzzo et al. (2014) and Scavuzzo (2013), we conducted several tests to measure the performance of our initial implementation. In particular, in these tests, we performed the following data migrations: *a)* we migrated a dataset stored in GAE Datastore DaaS according to eventual consistency semantics, towards Azure Tables DaaS; *b)* we migrated a dataset stored in Azure Tables DaaS according to strong consistency semantics, towards GAE Datastore DaaS. The adoption of two different semantics in the usage of the two DaaS therefore allowed us to experiment with data migration in both conditions.

The migration system was deployed on a Google Compute Engine (a IaaS platform) Virtual Machine (VM) – physically hosted by the Google data-center in Western-Europe

Table 1 Migrations preserving eventual consistency

	From GAE Datastore to Azure Tables			From Azure Tables to GAE Datastore
	dataset #a	dataset #b	dataset #c	dataset #a
Source size (MB)	16	64	512	-
# of Entities	36940	147758	1182062	36940
Migration time (s)	1098	4270	34111	13101
Entities throughput (ent/s)	33.643	34.604	34.653	2.820
Queued data (MB)	81.98	336.73	2709.80	93.50
Extraction&Conversion time (s)	31	120	768	24
Queued data throughput(KB/s)	2707.985	2873.446	3613.067	3989.513
Avg. %CPU usage	4.749	3.947	4.111	0.605

(WE) – with two virtual CPUs and 7.5GB of RAM, running a Linux-based operating system. Data were stored in the form of entities (both in GAE Datastore and in Azure Tables) whose average size is 754 bytes.

In Tables 1 and 2 we report the results obtained when performing the migration between these two DaaS. Tests were performed three times (tables report average values) because of the homogeneous behaviour found across all of the runs. Multiple data sets have been considered. *Source size* indicates the size in Megabytes of the entities we have migrated in that particular test (whose number is denoted with *# of Entities*); *Migration time* reports the time, in seconds, needed to complete that particular migration; *Entities throughput* shows the ratio between the number of migrated entities and the migration time; *Queued data* indicates the total size of the entities (in the meta-model representation format) that have been stored inside the queue during all the migration process; *Extraction and conversion time* shows the time needed to extract all the entities from the source DaaS, converting them into the meta-model representation, and to store them in the queue; *Queued data throughput*

Table 2 Migrations preserving strong consistency

	From Azure Tables to GAE Datastore			From GAE Datastore to Azure Tables		
	dataset #d	dataset #e	dataset #f	dataset #d	dataset #e	dataset #f
# of Entities	9235	36940	55410	9235	36940	55410
Migration time (s)	1402	5340	8599	275	1098	1645
Entities throughput (ent/s)	6.587	6.918	6.444	33.581	33.643	33.684
Queued data (MB)	22.40	89.61	134.41	22.97	93.50	140.33
Extraction and Conversion time (s)	10	30	41	8	31	45
Queued data throughput (KB/s)	2294.067	3058.627	3357.047	2940.16	3088.51	3193.29
Avg. %CPU usage	1.509	1.139	0.957	4.932	4.746	4.564

is the ratio of the two previous rows; finally, *avg. %CPU usage* reports the average CPU percentage measured in the VM hosting the migration system.

Test results, reported in the first three columns of Table 1, showed that we were able to migrate entities from GAE Datastore towards Azure Tables, preserving eventual consistency, at an average rate of 34 entities/s; as determined experimentally, this rate is independent from the dataset size.

In order to verify the reverse migration and to check the correctness of our migration process, we transferred data, already migrated, back to GAE Datastore. During these experiments we experienced several undocumented errors[20] thrown by the DaaS when migrating more than 70,000 entities towards GAE Datastore, under various working conditions, and with different data sets. For this reason, we report in the fourth column of Table 1 only the experiment concerning 36,940 entities. In this case we checked that the migration was conservative by verifying that the number of migrated entities was the same as in the counterpart experiment shown in the first column of Table 1 and that their content was identical to the one we migrated in this first test. The migration time, however, was increased by a factor of 11.93. This is because entities managed according to an eventual consistency policy (thus, belonging to different Partition Groups in the intermediate format) are required to have different *Ancestors* in Datastore (see Section 2.2). Thus, every write operation requires not only the creation of the entity in the database, but also the creation of a corresponding ancestor entity (this corresponds to two write operations, plus one read to check if that ancestor is already existing).

Table 2 reports the results achieved when migrating entities from Azure Tables to GAE Datastore, and viceversa, preserving strong consistency. Once again, the migration to GAE Datastore is rather slow because of the management of the ancestor path mechanism. More in detail, as data to be handled according to the strong consistency semantics in Datastore have to be connected to the same ancestor, the migration time towards GAE Datastore includes not only the write operations, but also all reads needed to retrieve the ancestor to be used for grouping the entities. As regards data migration back into Azure Tables, the results we obtained, in terms of throughput, are similar to the ones in Table 1, obtained when preserving eventual consistency. In this case, in fact, no computationally intensive operation is performed to maintain strong consistency in Azure Tables (it is just a matter of setting the same Partition Key when translating entities contained in the same meta-model Partition Group (Scavuzzo et al. 2014)). Also in these experiments we verified that the migration was conservative.

In both scenarios, the use of CPU by Hegira4Cloud was always negligible. However, the limited throughput we obtained in all cases proved that this initial prototype is not suitable to manage Big Data problems.

5 Hegira4Cloud v2: First Round of Improvement Through Action Research

As shown in the previous section, the experiments we conducted on Hegira4Cloud v1 highlighted several undocumented errors caused by Google Datastore. These were neither described in the documentation, nor fixable by means of the software development kit

[20] RemoteApiException: remote API call: unexpected HTTP response: 500

provided by Google. Furthermore, the experiment also highlighted low performance of our system when trying to migrate data between the two selected NoSQL DaaS; the reason of such behaviour was the scarce optimization of Hegira4Cloud v1.

Understanding the reasons of such behaviours and identifying their causes required the execution of numerous tests, given the lack of tools for design-level analysis and optimization of our framework (as we will discuss in Section 7.3). More specifically, we decided to address the problem of improving the performance of Hegira4Cloud through action research (approaches to perform action research are presented in Baskerville and Myers (2004) and O'Brien (1998)), exploring the solution space through the development of various experiments and associated prototypes. Given the inexplicable and undocumented behaviour we experienced while writing data on GAE Datastore, we decided to perform our experiments referring to the migration from Datastore to Azure Tables and not vice versa.

The cost of performing such experiments was rather high, since they have been conducted against two pay per use DaaS and it required a lot of effort and time to engineer the prototype.

In our space exploration approach we went through the following steps:

1. We started testing the read and write performance limits of the DaaS systems we were considering. This first analysis is described in Section 5.1 and brings to the conclusion that the performance of Hegira4Cloud v1 can be significantly improved.
2. Based on the previous consideration, we decided to check if the bandwidth between the clouds involved in the migration (Azure and Google in the experiments) was the limiting factor. We measured a good throughput when transferring data from Google to Azure (see Section 5.2), thus we concluded that bandwidth is not a bottleneck for our system.
3. Then we decided to introduce parallelism in the Hegira4Cloud software architecture to see if this could allow us to obtain some improvements. This experiment has implied a significant redesign of the system (see Section 5.3) and has led to a significant performance improvement.
4. However, despite the performance increase, we realized that Hegira4Cloud throughput was still far away from the DaaS reading and writing performance limits identified at step 1. Then, we decided to enhance the data serialization step and this led to a further performance improvement (see Section 5.4).
5. Finally, we evaluated the new version of Hegira4Cloud on an industrial case-study, by using a large dataset extracted from Twitter (see Section 5.5) and we assessed its exten-sibility developing new translators to support migration from/to Apache Cassandra and Apache HBase that, at the time of writing, are the most well-known NoSQL non-DaaS databases (Section 5.6).

In the following subsections, we extensively present each step of this action research process.

5.1 Testing the Read and Write DaaS Limits

The read and write throughput from/to the NoSQL databases under consideration can certainly be considered as an upper bound for any migration system. For this reason, our investigation started with an analysis of such throughput. We could not find any information on this in the documentation offered by GAE Datastore and Azure Tables. In Hill et al. (2010) authors measured Azure Tables performance by issuing CRUD (i.e., create, read, update, and delete) operations, but since those tests were performed some years ago, they do not necessarily reflect the current situation. Thus, we decided to perform our own experiments.

The focus of our analysis was on the write operations as the read throughput from the source DaaS (either Datastore and Tables), at a first analysis, does not appear to be a bottleneck for the migration system. In fact, based on the preliminary results we obtained in Tables 1 and 2, we can conclude that Hegira4Cloud v1 is able to read and process data at a quite high throughput, up to 1,539 entities/s (evaluated as the ratio between the number of migrated entities and the extraction and conversion time).

Since we decided to focus on Azure Tables as the target NoSQL, we concentrated our analysis on this DaaS and we developed a tool which allows us to measure the maximum number of parallel writes an application is able to perform in the same Azure Table account. The reader should note that, while currently, several NoSQLs benchmarking tools are available (e.g., Yahoo! Cloud System Benchmark (YCSB), Netflix Data Benchmark (NDBench), etc.), when our simple benchmarking tool was developed (January 2013) YCSB did not support Azure Tables and NDBench did not exist.

In the experiments, we considered entities having different average size (754 Byte, 1KB and 4KB), trying to understand how entity size impacts Azure Tables write speed. For the tests we used an Azure VM with the following configuration: Ubuntu Server 12.04, located in Microsoft WE data-center, with 4 CPU and 7 GB of RAM. Moreover, we used *log4j library* to compute tests duration and a *custom library* to measure the size of objects stored in the queue and the size of Azure Tables entities (Chauhan 2012). Table 3 summarizes the test results. For each run we transferred a constant number of entities (147,758 entities) and we varied the following parameters:

- The Entity Average Size (*EAS*), as well as the Entities Total Size (*ETS*).
- The number of consumer threads, *#threads*, writing in parallel to Azure Tables.

As a result of the tests we collected the following output performance metrics:

1. The overall time *Tt* required for writing entities in Azure Tables.

Table 3 Azure Tables entities throughput

EAS	ETS(MB)	#threads	Tt (s)	X (ent/s)	X (KB/s)
754 Byte	106	10	217.3	680.0	499.5
754 Byte	106	20	121.3	1218.1	894.8
754 Byte	106	40	102.3	1444.4	1061.0
754 Byte	106	60	122.0	1211.1	889.7
754 Byte	106	192	127.3	1160.7	852.7
1KB	152	10	230.0	642.4	676.7
1KB	152	20	156.0	947.2	997.7
1KB	152	40	129.0	1145.4	1206.6
4KB	580	10	244.0	605.6	2434.1
4KB	580	20	148.0	998.4	4013.0
4KB	580	40	139.0	1063.0	4272.8
4KB	580	60	120.0	1231.3	4949.3
4KB	580	192	116.6	1267.2	5093.7

2. The maximum throughput X achieved for data transfer. We report it both in terms of entities/s and in KB/s values.

Each of the values in the table is the average obtained from three runs with the same inputs.

We found that Azure Tables maximum write throughput varied between 600 and 1,444 entities/s, depending on the number of threads writing in parallel on it. While it seems we reached a limit in terms of writeable entities per second, the throughput measured in KB/s kept increasing with the entities average size, proving that the database limits the number of operations per second and not the throughput in terms of bytes per second. A hypothesis explaining such behaviour could be that Microsoft (and other multi-tenant DaaS vendors), in order to provide acceptable response times, limits the number of users' requests.

The values obtained with this experiment, if compared with the results we obtained with our preliminary version of Hegira4Cloud in Section 4 (up to 34.6 entities/s), show that Azure as a target NoSQL DaaS is not the system bottleneck and the limited throughput we obtained with Hegira4Cloud v1 is due to some other cause. Moreover, when comparing this writing throughput with the one we obtained for the read operations from both Tables and Datastore, we can conclude that they are aligned and therefore we can confirm that the reading throughput is not a problem either.

5.2 Checking if the Bandwidth is a System Bottleneck

The next step towards the identification of the bottleneck consists in measuring the maximum network bandwidth available for data migration.

For these tests, we deployed two VMs on Azure and one on Google Compute Engine. More specifically, the Azure VMs had the following configuration: Ubuntu Server 12.04, located in the same Microsoft WE data center (not in the same virtual network), with 4 CPU cores and 7 GB RAM. The Google VM had the following characteristics: Debian 7, hosted in Google WE data-center, with 2 virtual CPUs and 7.5GB of RAM.

On each of these VMs we installed the *iperf* tool,[21] able to measure the maximum bandwidth on IP networks among two VM instances.

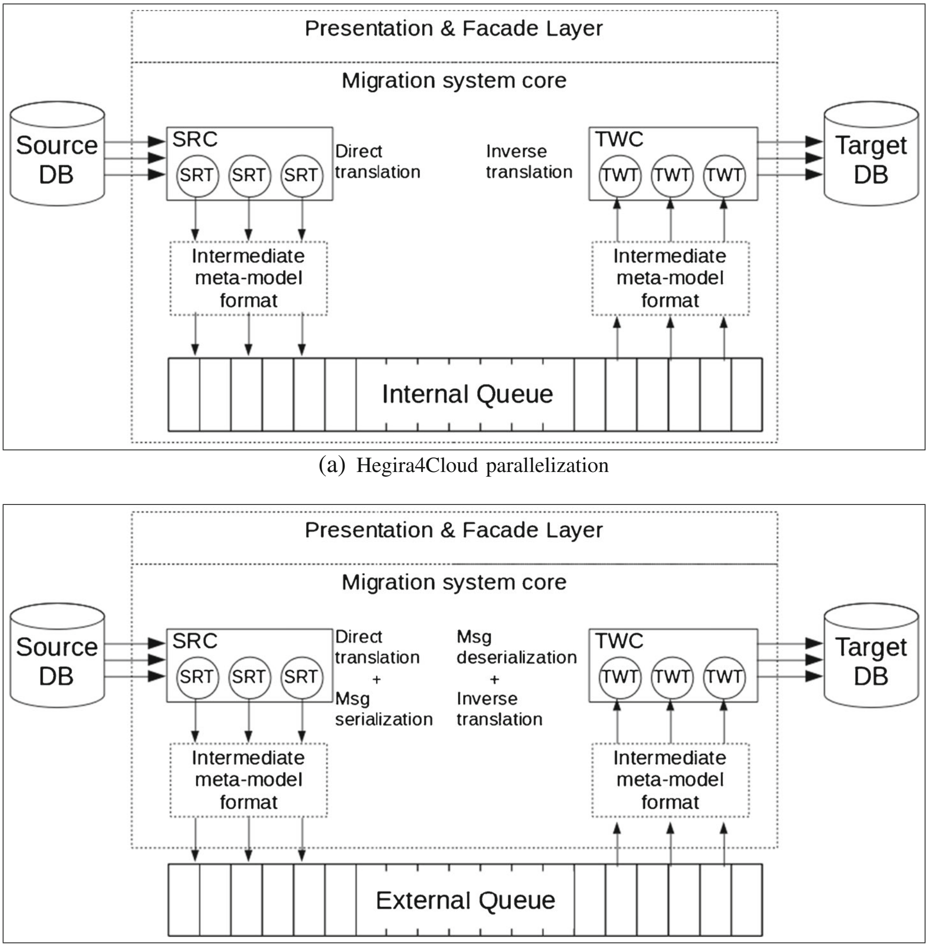
With a first test we measured the maximum incoming bandwidth from the Google VM to one of the Azure VMs, trying to understand if the bandwidth between the cloud data centers could be the bottleneck. We found that the maximum incoming bandwidth (average of five runs) was 418 Mb/s, which is certainly much larger than the queued data throughput (3,112 KB/s on average, i.e., 24.3Mb/s) shown in Table 1. Thus, we can safely state that data entering in the migration system is not limited by the network connectivity.

In order to understand if the maximum number of entities that can be written to Azure Tables (measured in previous Section 5.1) could be limited by the bandwidth between the migration system (deployed on Azure) and Azure Tables, we performed a second test, in which we measured the maximum bandwidth between two Azure VMs, in different local networks. The bandwidth we measured (average of five runs) was 712 Mb/s. By comparing this value with the maximum throughput we obtained in Section 5.1 (5093.7KB/s, i.e., 39.8Mb/s), we can state that also the outgoing VM bandwidth is not a bottleneck for the migration system.

[21] https://github.com/esnet/iperf

5.3 Experimenting with Parallelism: the New Hegira4Cloud Architecture

A typical way to improve performance is to introduce some degree of parallelization in the operations to be executed. In the case of Hegira4Cloud, the main operations concern reading from the source NoSQL database and writing into the target one. As shown in Fig. 2, the SRC process is in charge of reading the data to be migrated from the source database and the TWC process is responsible for writing the corresponding data in the target database. Thus, the parallelization may consist in the definition of various parallel threads within each of the two processes. As shown in Fig. 3a, some of these threads, which we call *Source Reading Threads* (SRTs), read data from the origin NoSQL database, transform them in the intermediate format, and put them in the internal queue. Some others, which we call *Target Writing Threads* (TWTs), get data from the internal queue, transform them into the target format and write them in the destination NoSQL database.


Presentation & Facade Layer
Migration system core
Source DB
SRC
SRT
SRT
SRT
Direct translation
Inverse translation
TWC
TWT
TWT
TWT
Target DB
Intermediate meta-model format
Intermediate meta-model format
Internal Queue
Presentation & Facade Layer
Migration system core
Source DB
SRC
SRT
SRT
SRT
Direct translation + Msg serialization
Msg deserialization + Inverse translation
TWC
TWT
TWT
TWT
Target DB
Intermediate meta-model format
Intermediate meta-model format
External Queue

(a) Hegira4Cloud parallelization

(b) Hegira4Cloud v2

Fig. 3 Hegira4Cloud architecture evolution

Analyzing the APIs offered by NoSQL databases, we realized that not all of them (e.g., GAE Datastore) provide an absolute pointer to the data that is being read. This means that building many parallel SRTs without duplication or data loss requires to somewhat partition the data in the source database and to employ a specific synchronization protocol between the SRTs to process data partitions separately. Since, as discussed in Section 5.1, the throughput of read operations did not seem to be an issue, we decided not to parallelize reading operations in Hegira4Cloud v2. So, we decided to use a single SRT and we focused on the parallelization of TWTs. This choice has been reconsidered in Hegira4Cloud v3 (see Section 6) when we partitioned data to improve the fault-tolerance of the data migration process.

TWTs parallelization requires addressing the following problems:

– Data should be properly split among TWTs: we should avoid an incorrect (i.e., not unique) data dispatching to TWTs, since this may cause data duplication into the target NoSQL database introducing, possibly, errors at the application layer.
– In case of system failure, data should not be lost, and should be properly migrated on system recovery.

Since these problems are typically addressed by message queues, usually provided as external middlewares, we decided to introduce one of these in Hegira4Cloud, thus moving from a monolithic architecture to a distributed one. In particular, the usage of the external queuing mechanism allows us to split SRC and TWC components into two separate systems, which are not forced to reside in the same Java Virtual Machine(JVM) and can be distributed over different servers (see Fig. 3b), thus allowing us to horizontally scale Hegira4Cloud. In particular, we adopted the *RabbitMQ*[22] open-source message queue implementation because of its compliance with the Advanced Message Queuing Protocol (AMPQ) standard (Godfrey et al. 2014) and its fault-tolerance properties.

The number of TWTs to be used for migration can be chosen by the user by means of the Hegira4Cloud REST API (depicted in Fig. 3b as Presentation & Facade Layer). After initialization, each TWT opens a connection to the destination database and runs in a thread-pool, waiting for a message (i.e., a metamodel entity) to be processed. Messages are assigned to TWTs by the RabbitMQ broker using the Round-Robin algorithm; hence, every TWT gets, on average, the same number of messages (thus preventing starvation phenomena from happening).

An acknowledgement mechanism[23] is used to prevent losses of messages already dispatched to a TWT (which can occur when, e.g., a TWT gets an error from the target NoSQL database and the insert operation cannot be completed).

To avoid message losses caused by the failure of the entire migration system or of RabbitMQ, Hegira4Cloud v2 configures RabbitMQ to persist queued data on disk; we expect this mechanism to be less efficient than the default in-memory storage, but, in the event of queue faults, we would be able to recover the data already stored in the queue without reading them again from the source database.

The implementation of SRT and TWT operations offered by Hegira4Cloud v2 is almost unchanged compared to the implementation of the corresponding operations within Hegira4Cloud v1, except for the fact that:

[22] http://www.rabbitmq.com/

[23] https://www.rabbitmq.com/confirms.html

1. As shown in Fig. 3b each entity in the intermediate format is serialized by the SRT before being sent to the external queue and deserialized by the TWTs before it is translated in the target format. Hence, Hegira4Cloud v2 introduces another (de)serialization step in addition to the one concerning each single property value field (see Section 3). This additional serialization step cannot be avoided since RabbitMQ (as well as other external queuing messaging systems) can only store messages in binary format.
2. The SRC and the TWC can be executed in distributed mode, i.e., as separate systems, on different JVMs.

In order to evaluate the performance of the new architecture, several preliminary tests were conducted aiming at verifying that this architecture was behaving as expected (a more extensive test on Hegira4Cloud v2, against an industrial case study, has been performed at the end of the whole optimization process and is presented in Section 5.5).

The migration system was deployed according to a simple, standalone configuration, i.e., the SRC, the TWC and a Single RabbitMQ broker were all installed within an Azure VM (with the same characteristics of Section 5.1). Tests transferred data from GAE Datastore to Azure Tables. In particular, as for the experiments in Section 5.1, we migrated 147,758 entities, corresponding to 64 MB, originally persisted on GAE Datastore with an eventual consistent semantics. After the migration, the measured size of the target Azure Tables entities was 106MB, because of the additional information we store to preserve data types, secondary indexes and read consistency policies. Data stored inside the message queue were even bigger 336.73MB, because of: *i)* the overhead introduced by the intermediate format in order to preserve mainly data types and secondary indexes; *ii)* the overhead introduced by the Java serialization protocol. Tests results are reported in Table 4. Besides the number of entities to be transferred (147,758 in these experiments), the other configuration parameter is the number of TWTs (*#TWTs*), writing in parallel to Azure Tables. The tests provide the following output performance metrics:

1. Overall time required for migration, *MT*.
2. Standard deviation of the migration time from the average value, *STDM*.
3. Throughput in terms of entities exiting from the system per second (*EET*) and in terms of KB/s (*EDT*).
4. Time needed for data reads (from the source database), conversion to/from the intermediate format and writes into queue, *ECT*.
5. Throughput in terms of KB/s stored in the queue, *QDT*.
6. Average CPU utilization of the VM hosting Hegira4Cloud, *CPU*.

For each number of TWTs, three different runs of the same test were performed (results in Table 4 are average values) because of the homogeneous behaviour found across all of the runs (i.e., the standard deviation from the migration time average value was always below 2 %).

Table 4 Data Migration from GAE Datastore to Azure Tables

#TWTs	MT(s)	STDM	EET(ent/s)	EDT(KB/s)	ECT(s)	QDT(KB/s)	CPU
10	426	7.37	346.85	254.80	117	2947.11	24.10 %
20	321	5.25	460.30	338.14	158	2182.35	33.06 %
40	306	4.79	482.87	354.72	173	1993.13	35.56 %
60	304	3.81	486.05	357.05	176	1959.16	36.23 %

As in Section 5.1, we used *log4j* for computing tests duration and our custom library for measuring entities size. Moreover, we used the *sysstat package* to measure CPU usage and the time required to create the objects stored in the queue in the intermediate format. Finally, we used *GAE Datastore Statistics* to evaluate the average size of the migrated entities.

From the results, we can notice that, with the adoption of a single SRT and multiple TWTs approach, the overall migration time is decreased up to 14 times, with respect to our initial version, at the cost of a greater CPU utilization (up to 9.2 times) and of an increased *Extraction and conversion time* (up to 1.5 times, mainly due to the (de)serialization process). It seems also that we reached the maximum number of threads useful for maximizing entities throughput, because, with the current setup, we cannot notice any significant throughput improvement using 60 TWTs instead of 40 TWTs. Notwithstanding, this threshold can vary according to the underlying hardware architecture running the TWC.

However, there is still space for improvement. In fact, even though the migration system performance has significantly improved, the maximum throughput achieved is 2.5 times lower than the ideal maximum according to the results of Section 5.1. As stated in Section 5.2, the network bandwidth is not a bottleneck for the migration system. Moreover, by analyzing the tests conducted by others[24] on RabbitMQ with a single broker deployment, we can conclude that this queue can handle a message rate that is higher than the one we have achieved (up to 10,000 messages per second). Hence, we can safely assert that the queue does not limit the whole migration system performance.

Excluding the causes above, we assumed that the double serialization and de-serialization performed by the migration system to preserve database data types and to store data in the queue, may have caused a significant performance drop. This assumption is investigated in the following section.

5.4 Optimizing the Serialization Process

The Hegira4Cloud version tested in the previous section adopts the Java standard serialization mechanism (to encode/decode metamodel data and to preserve data types during the migration), which, according to the serialization benchmarks published in 'https://github.com/eishay/jvm-serializers/wiki', is significantly slower than other solutions and generates, on average, larger serialized data.

Based on these benchmarking data results, we decided to change the (de)serialization mechanism and to adopt the one included in Apache Thrift,[25] which, according to the same benchmarks, appeared to be one of the most performant. Thrift is a software framework for scalable cross-language services development, originally developed at Facebook and then open-sourced as an Apache project. In contrast with other solutions like Apache Avro[26] or Protocol Buffers,[27] Thrift, at the time of writing the system, also supported complex data types (e.g., HashMaps, Lists and Sets) which are extensively used by our intermediate format implementation. Apache Thrift uses an *Interface Definition Language (IDL)*, which generates the source code implementing the actual (de)serialization process. This

[24] http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/

[25] https://thrift.apache.org/.

[26] http://avro.apache.org/

[27] https://developers.google.com/protocol-buffers/

Table 5 Data migration from GAE Datastore to Azure Tables using Apache Thrift

#TWTs	MT (s)	STDM	EET(ent/s)	EDT(KB/s)	CPU
10	294	3.41	502.58	222.91	58.09 %
20	237	3.87	623.45	276.52	74.66 %
40	168	2.17	879.51	390.10	74.96 %
60	169	0.71	874.31	387.79	75.29 %

implies that the format of data to be (de)serialized is defined at design-time and cannot change dynamically. For this reason, Thrift cannot be used for that part of our process that (de)serializes property value fields for preserving data types. Nevertheless, it is suitable for the other serialization step that is needed to store data in the RabbitMQ queue. To adopt it, we expressed the Meta-model structure by means of Thrift IDL and we generated the corresponding Java source code which replaces the old Meta-model classes. Then, the standard Java (de)serialization calls, performed to interact with the queue, were replaced by Thrift (de)serialization calls.

Using the same environmental setup of previous sections, with the new implementation, we were able to achieve the results reported in Table 5: the migration system is now able to migrate up to 879 entities/s and, hence, it is 1.8 times faster than the old configuration described in Section 5.3 and 25.4 times faster than the first version of Hegira4Cloud described in Section 4. The high CPU usage value reported is due to an initial overhead of Hegira4Cloud v2, which stabilizes after 120 seconds and, since we moved a relatively small amount of entities in a short period of time (i.e., 294 seconds, at maximum), this peak highly contributes on the CPU usage. Overall, the migration system is not yet as fast as possible (in fact, it is 1.6 times slower than the upper bounds determined in Section 5.1) because of the overhead introduced to preserve different data types, secondary indexes, and consistency polices. Notwithstanding, we think we have achieved a reasonable trade-off between performance and the inevitable overhead introduced to meet Hegira4Cloud requirements (in particular, correctness and failure recovery).

5.5 Hegira4Cloud v2 Evaluation on a Large Dataset

In this section we evaluate the performance of Hegira4Cloud v2 using a large data set extracted from Twitter. In particular, we stored into GAE Datastore 106,937,997 tweets publicly available at ArchiveTeam (2012) and then we ran Hegira4Cloud to migrate them into Azure Tables. The purpose of the experiment is to check if this new version is able to perform the migration with an acceptable overhead and without introducing errors directly due to the migration process.

5.5.1 Experimental Setup

As mentioned before, our data set was composed of 106,937,997 tweets. Each tweet, in addition to the 140 characters long message, contains also details about the user, creation date, geospatial information, etc. Each tweet was stored in GAE Datastore as a single entity, containing a variable number of properties (with different data types). On average, each tweet size on Datastore was 3.05KB. The total entities size was, thus, 311GB. The migration system, together with RabbitMQ server, was deployed inside an Azure VM configured as described in Section 5.1.

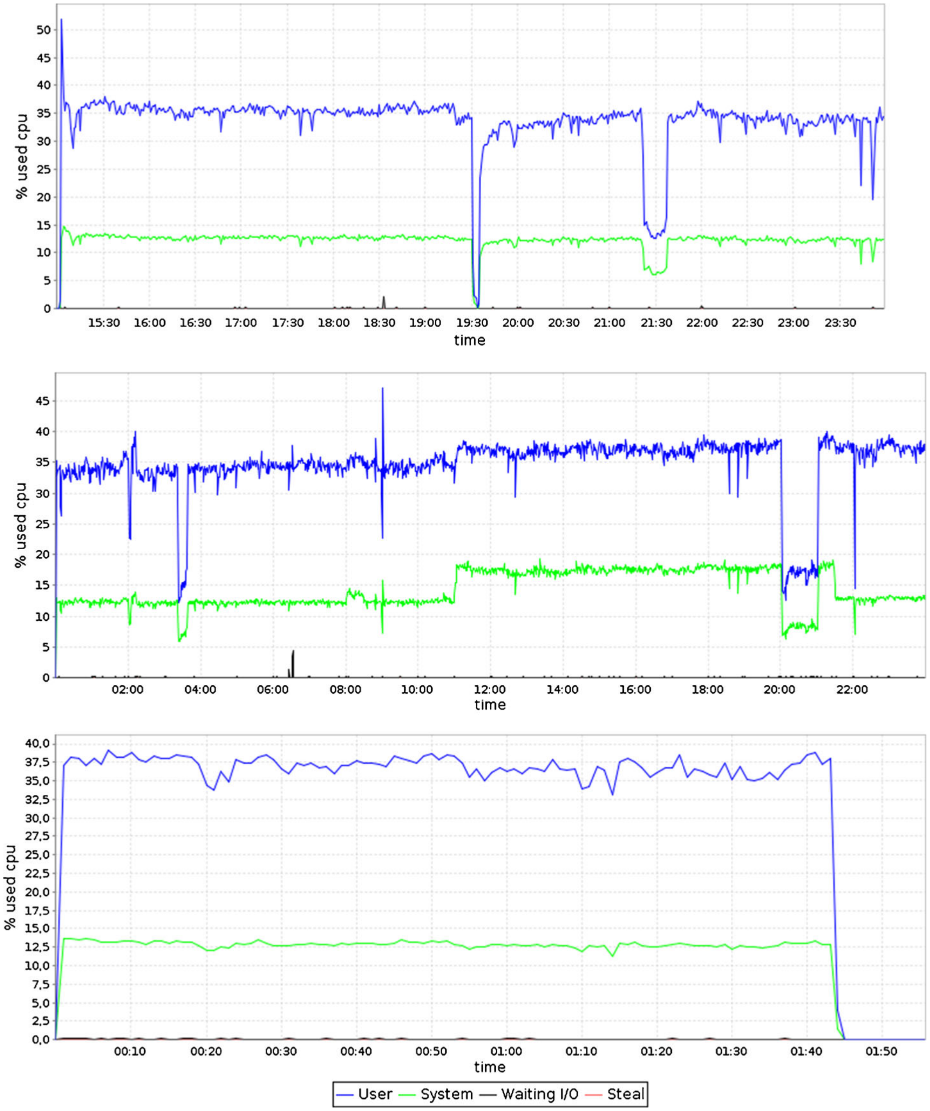


Fig. 4 CPU usage

5.5.2 *Results*

We migrated the tweets using 40 TWTs to write data in parallel on Azure Tables. We chose to use 40 TWTs since, as discussed in Section 5.3, we did not experience any improvement experimenting with Azure Tables and a larger number of TWTs.

The overall migration took almost 34 hours and a half (124,867 seconds). The main system metrics we measured were the CPU utilization of the hosting VM and the bandwidth incoming to and outgoing from Hegira4Cloud. They are reported in Figs. 4 and 5. For readability, each figure shows three diagrams that split the time axis in three parts. We started

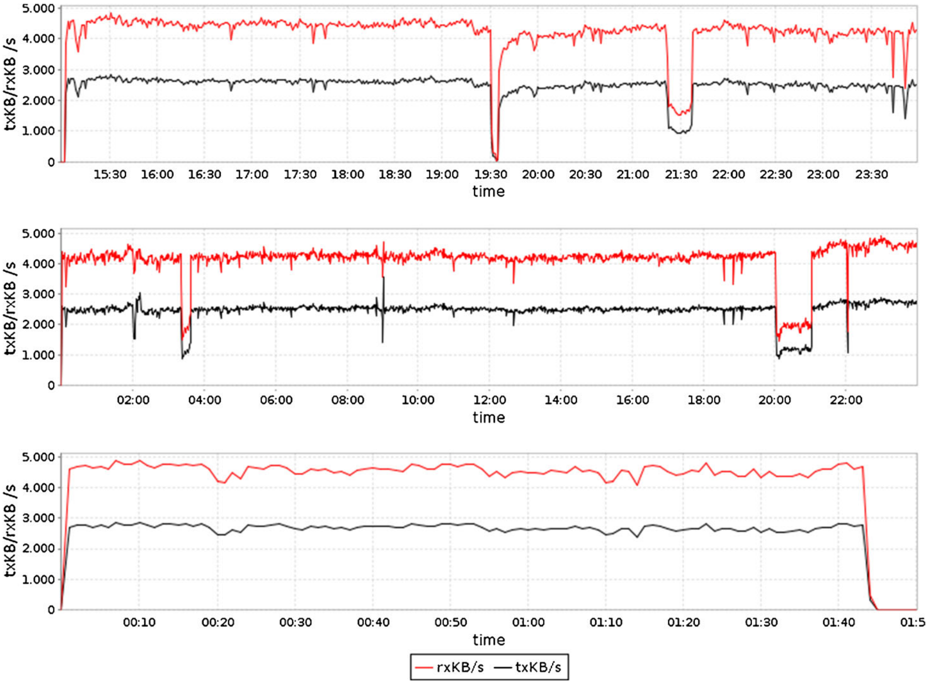


Fig. 5 Incoming and outgoing bandwidth

the migration at 15.00 CET of the first day (see the first diagram of the two figures) and it finished at 01.45 CET of the day after (third diagram). The average CPU utilization was 49.87 %, whereas the RAM usage, on average, was 45 %. The initial Hegira4Cloud overhead on CPU, as well as its stabilization, is shown in the first diagram of Fig. 4. Incoming (rxKB/s) and outgoing (txKB/s) bandwidth was almost constant for the whole period, with the exception of some slow down, lasting from few minutes up to one hour, and some sudden spikes. The overall migration throughput was 856.41 entities/s, and we were able to migrate entities at an average rate of 2,611.63 KB/s. Comparing these values with those of Table 5, we notice a similar throughput in terms of entities/s, instead, in terms of KB/s, the tweets migration throughput is higher by a 6.69x factor; this is due to the fact that, on average, entities corresponding to tweets have a larger size compared to those migrated in the previous section (the same behaviour also occurred in tests reported in Section 5.1).

Finally, we also monitored the total cost incurred by the migration process. Referring to the experimental setup described in Section 5.5.1, we considered the factors for cost computation:

1. *GAE*
   - Datastore read operations are performed to extract entities from this DaaS.
   - Datastore output bandwidth is computed to transfer Datastore entities toward Hegira4Cloud.
   - GAE Front-end Instance hours as requests to the Datastore can only be issued throughout GAE front-end instances; hence, this cost parameter refers to the

Table 6 Migration costs

		Price per unit (*)	Resource usage
GAE	Datastore read ops	0.6 $/Mops	109.35 Mops
	Datastore bandwidth out	0.12 $/GB	321 GB
	GAE instance hours	0.05 $/h	165.04 h
Microsoft Azure	VM compute hours	0.24 $/h	35 h
	VM bandwidth out	0.19 $/h	311 GB
	Tables storage transactions	0.05 $/Mtx	107 Mtx

(*) on June 2014.

instances, brought up by GAE, upon read requests (through RemoteApi) issued by Hegira4Cloud.

2. *Microsoft Azure*

   – Virtual Machine compute hours, needed to perform the whole migration process.
   – Virtual Machine output bandwidth needed to transfer the entities into the target DaaS.
   – Azure Tables storage transactions needed to persist entities into the target DaaS.

Costs are reported in Table 6, along with the unitary price and the amount of resource usage. Notice that, we excluded the storage costs (both of the source and the target databases) on purpose, since they do not represent a migration cost and they depend on how long the data are stored in the databases. Furthermore, Microsoft Azure does not charge the user for input bandwidth. All of this considered, we can conclude that the cost of migrating our 106,937,997 tweets from Datastore to Tables has been 0.582$/GB, 181$ in total.[28] This value depends on the data to be migrated, since the number of write operations and output bandwidth may change depending on the data types adopted and on the presence of secondary indexes.

5.5.3 Discussion

From the analysis of results we can conclude that Hegira4Cloud v2 is suitable to handle and process huge quantities of data with a very high throughput. The migration system could not reach the maximum write throughput of Azure Tables (see Section 5.1), even though it is of the same order of magnitude, because of the overhead introduced by the double (de)serialization process, which indeed, is required to preserve source and target database characteristics and to exploit message queue fault tolerance and recovery features in case of system crashes. The other reason that limited the throughput was the performance degradation occurred retrieving and persisting data from/to the databases, probably caused by their multi-tenancy. We noticed such degradation by monitoring the system during the tests execution. We did not notice it in the tests of Section 5.1, but it emerged in the final tests, as, in this case, we could observe the interaction with the DaaS for a significantly longer period of time.

[28] The total cost can be obtained by summing, per each voice of cost in Table 6, the value obtained by respectively multiplying the "Price per unit" with the "Resource usage".

The improvement we achieved in Hegira4Cloud v2 comes at the cost of an increased CPU and RAM usage; in fact, we noticed much higher values, for these parameters, with respect to tests of Sections 4 and 5.3. Furthermore, we noticed a correlation between CPU usage and network bandwidth, which is not surprising since the (de)serialization process heavily impacts on CPU usage; i.e., when the system retrieves/stores fewer data, the (de)serialization processes also slow down and the CPU usage decreases consequently.

As in the previous experiments, we ran the data migration of the Tweets dataset three times. Unfortunately, only a single run was successful. By inspecting the application logs we found that the reason was an undocumented and unrecoverable error (HTTP 500) in Google Datastore that suddenly stopped read operations and prevented the established connection from finishing to read the data to be migrated. Moreover, since Google Datastore does not guarantee a unique pointer to the data after a connection is terminated, client applications cannot consistently recover interrupted read operations, i.e., given two separate connections to Google Datastore executing the same query, it is not guaranteed that they will receive the data in the same order. Hence, in our case, simply restarting the failed data migrations, would have caused inconsistencies of the migrated data in the target database. In Section 6 we address this problem and we provide a solution to it.

5.6 Evaluating the Extensibility of Hegira4Cloud

In order to assess the extensibility of Hegira4Cloud, we developed new translators to add support to Apache Cassandra and Apache HBase. The implementation of the Cassandra translators was flawless and quite immediate, thus allowing us to quickly add support for data migration from/to Cassandra.

In the specific case of Apache HBase, we experienced several challenges in the development of the direct translator (i.e., the one that converts data from HBase data model into the intermediate metamodel format). In fact, HBase, differently from any other database we have supported so far, does not store a schema of the entities it contains, but it simply stores data in binary format. This implies that any application that requires to read data from HBase must already know the schema of the data contained in the database in order to interpret them. This lack of schema management capabilities of HBase limits Hegira4Cloud data migration approach, in that it does not make it possible to automatically derive schemas from the data read from HBase. Hence, at the moment, HBase is supported only as a target database (i.e., data can only be migrated to it). We plan to support data migration from HBase as a future work. Hegira4Cloud v2, together with the new translators for Cassandra and HBase, was used in a real use-case scenario (Picioroaga and Nechifor 2014; Ferry et al. 2015) to migrate data generated from sensors, and stored into Cassandra, towards an HBase cluster. As a future work, we plan to thoroughly evaluate Hegira4Cloud performance when migrating data across these two on-premise NoSQL databases.

6 Hegira4Cloud v3: Addressing Fault-tolerance

As explained in Section 5.5.3, even though Hegira4Cloud v2 was designed to tolerate faults occurring in any of its internal components, still it was not able to complete two of the three experiments of tweets migration because of an undocumented and unrecoverable error in GAE Datastore. In general, any time the source database involved in the migration fails in the interaction with Hegira4Cloud v2, the system is not able to recover and to continue the migration from the point it has been stopped. The only possible solution is to start the

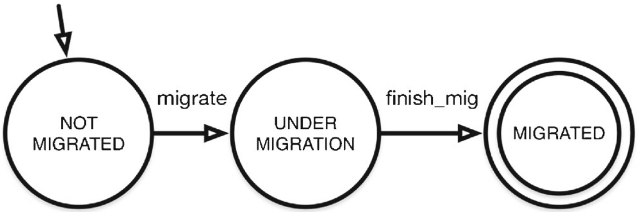


Fig. 6 Final state machine representing a VDP migration status

migration again from scratch, thus wasting time, bandwidth and resources that, as we have discussed in the previous sections, can be significant in the case we are handling very large datasets. The problem does not apply when the failure occurs in the interaction with the target database as in this case Hegira4Cloud v2 is already able to keep trying contacting the target database without losing the data to be stored.

Thus, to make Hegira4Cloud fully fault tolerant, we needed a way to consistently keep track of already migrated data (e.g., by logging), so that, in case of fault, we could be able to retrieve only the data that was not processed before the fault occurred. Such an approach should also be as efficient as possible in order not to add significant overhead to the data migration process. For these reasons, we have developed the third version of the system that is able to consistently recover read operations from any NoSQL database. The technique we have adopted virtually partitions data stored in the source database in so-called *Virtual Data Partitions* (VDPs) and assigns them a unique identifier so as to retrieve them in an ordered and unambiguous way, independently from the database being used. Data are assigned to a given VDP based on their primary key,[29] hence we assume that primary keys are assigned by the applications writing in the source database in sequential order and that the last assigned sequential primary key is known by the migration system. Similarly to the Linux paging technique, this approach can associate stored entities to VDPs and viceversa.

When a data migration process is started for the first time, the migration system creates a snapshot of the data currently stored in the source database. The snapshot consists in a list of VDPs computed using the last assigned sequential primary key and a user-provided VDP size.[30] Additionally, each VDP is associated with a status (represented in Fig. 6 by a finite state machine, in short, FSM) that can assume three values: NOT MIGRATED when the snapshot is first computed; UNDER MIGRATION when the SRC starts to retrieve the partition from the source database; MIGRATED when the TWC completes the storage of the partition in the target database.

To provide additional fault-tolerance guarantees, the snapshot and the last assigned sequantial id are stored inside a shared, replicated and fault-tolerant status-log, namely Apache ZooKeeper[31] (Hunt et al.2010a); by doing so, both the SRC and the TWC can safely and concurrently access the migration status. Further details, as well as the new architecture are reported in Scavuzzo et al. (2016). After the snapshot computation is over, the SRC extracts one VDP at a time from the source database and sets its status to UNDER MIGRATION. In parallel, after the TWC migrates a whole VDP, it updates its status to MIGRATED.

[29] (or any other indexable property if allowed by the database)

[30] Data migration users are able to choose the size of the VDPs before actually starting the migration; by doing so, users are able to trade data migration logging granularity for performance.

[31] http://zookeeper.apache.org

A data migration is terminated after all VDPs are migrated. By adopting this approach, we are able to instrument Hegira4Cloud for fault-tolerance with any NoSQL database. In fact, if at any point in time Hegira4Cloud v3 should stop (e.g., an error occurs) the migration can start again from the last migrated partition, thus avoiding transferring the same data twice. As reported in Scavuzzo et al. (2016), after an interrupted data migration is manually restarted, it takes less than 1 second for Hegira4Cloud to consistently recover a data migration process, independently from the source dataset size.

Consequently, we are able to reduce both the total migration time and, therefore, the migration costs, even in the event of faults. In future works we will conduct additional experiments to test whether this new fault-tolerant data migration approach has similar performance with respect to the standard data migration supported by Hegira4Cloud v2.

Alternative approaches to VDPs were considered, like, e.g., logging the status of every single entity under migration, but this approach would have caused data migration performance to drop significantly because of the additional writes to the log.

7 Open Challenges

At the time of the Hegira4Cloud conception and of the preliminary tests with our first prototype, we were convinced that we would have been able to move soon toward our real objectives, which are supporting on-the-fly synchronization between the source and the target DaaS and incorporating in the migration/synchronization framework many new NoSQL databases and DaaS.

Instead, we faced *the problem of properly interacting with the NoSQL DaaS we were using*. In fact, we realized that Azure Tables and GAE Datastore, similarly to other DaaS, were missing detailed information of the behaviour of their APIs. Thus, we had to derive experimentally the read/write times. We had also to experiment with parallel writes in order to understand the proper number of writing threads. Both providers we considered do not offer a predictable service, i.e., the number of entities that it is possible to read and write changes over time significantly and the providers do not guarantee any SLA. Research on SLA has developed proper languages and ontologies, but these approaches are not fully adopted yet.[32] Moreover, particularly with GAE Datastore, we experienced several random down-times which prevented us from successfully reading/writing entities from it; we noticed that the down-times were not correlated with our dataset or with a particular day or hour during which we performed our migration task. Thus, this led us to the conclusion that *Hegira4Cloud and, in general, any application that should be interfaced with DaaS should be designed to tolerate network faults, random database down-times and no performance guarantees*.

All *the experiments that we performed to understand the behaviour of such services implied a cost overhead* (even though not a large amount, i.e., almost 800$). This, in our case, was acceptable as we had research grants that could partially cover such expenses. However, this may not be a viable solution for a company willing to investigate if a certain DaaS is suitable for a specific Big Data problem.

We had to proceed with our development work using an expensive and time consuming trial and error approach that has required at least three iterations before reaching satisfiable

[32] Service Level Agreement Legal and Open Model (SLALOM) European Project – http://slalom-project.eu

behaviour. An important part of our engineering work has been based on exploiting parallelization to improve performance. Of course, this implies taking care of synchronization and fault tolerance issues and identifying the proper upper bound for the number of parallel threads/processes. Once again, for the kind of application we developed, all these aspects are greatly influenced by the characteristics of the used NoSQL databases (either DaaS or non-DaaS). The adoption of a message queue, as usual, has been certainly beneficial for decoupling SRTs and TWTs and for managing separation of concerns. By migrating large quantities of data and thus traversing different network infrastructures, we could measure a significant impact of the network latency over the total migration time. Furthermore, we could analyse the fluctuation in the CPU usage caused by sudden network latency degradation/improvement. Hence, in order to minimize the total migration time and reduce the CPU overhead as much as possible, we introduced a widely used and performant binary serialization framework, Apache Thrift, which was effective in reducing the size of the data sent over the distributed queuing system and the CPU overhead. Finally, to tolerate source databases unrecoverable errors, we introduced the virtual data partitioning approach that allows Hegira4Cloud to consistently recover a data migration to the last correct status, thus reducing the total migration time and costs in case of faults.

While the above issues are clearly related to our specific DI system, still we think that we can generalize our findings in terms of the following issues:

- There is a lack of well-documented DaaS. Moreover, DaaS are not based on properly defined SLAs.
- At the design level, as discussed in Section 2.1, there are frameworks that support parallel computations and integration with different data storage systems, but these are not flexible enough to cope with all possible application-specific aspects of a DI system.
- The literature offers some high level design guidelines for DI applications (see Section 2.1) as well as some tools to support modelling and QoS analysis/simulation of complex architectures, however, the available tools are not yet suitable to support DI systems that, still, have to go through the complex trial and error approach that we experimented in our work.

In the next subsections we provide more details on the above issues.

7.1 DaaS and SLA

When a NoSQL is offered as a DaaS, its internal architecture is opaque to the end-user, who has little or no knowledge of how the database actually handles data, unless a specific documentation is provided. Several elements contribute to their unpredictability:

- the adopted replication strategies that may differ from DaaS to DaaS: different data replication strategies can slow down queries response time;
- the relative locations of clients and DaaS nodes: geographical distribution of the DaaS nodes is crucial to estimate the response times;
- the adopted data storage management that may vary from DaaS to DaaS: the type of physical devices (e.g., spinning disk drives, solid-state drives, non-volatile memory, etc.) used to actually store the data, as well as the technique used to write data in the database (e.g., in-place updates, log-structuring storage) greatly influence read and write throughputs.

- the DaaS multi-tenancy (i.e., the fact that they are offered to multiple users): if no SLA is enforced by the provider, query response time may suffer during peak load;
- the DaaS network and nodes load.

In general, we can state that DaaS do not provide clear Service Level Agreements (SLAs), hence the behaviour of DI applications that rely on them may be highly unpredictable and require the execution of extensive experiments. Just to give an idea on the impact of DaaS characteristics on DI applications, the reader should consider that just a few slow data accesses per request can significantly raise response times, making the whole application less usable (Stewart et al. 2013) (e.g. for an e-commerce site, delays exceeding 100ms decrease total revenue by 1 %, which can cost more than an 11 % increase in compute costs). In the research field there are approaches to address SLA problems in databases. For instance, Terry et al. (2013) present a partitioned, geo-replicated and key-value data store which allows applications to choose consistency and latency priorities throughout consistency-based SLAs. In Das et al. (2013) authors propose an approach to achieve performance isolation in a multi-tenant relational DaaS, thus providing good performance assurance. However, to the best of our knowledge, there is just one commercial DaaS, Azure DocumentDB,[33] which ensures four predictable performance levels, allowing the user to choose an SLA in terms of consistency/latency.

7.2 Frameworks to Support Parallel Computation

In Section 2.1 we have briefly presented some of the most well-known frameworks for parallel computation, i.e., Hadoop, Spark, and Flink. They could have been exploited in the development of Hegira4Cloud, but they would have not addressed all our problems. The usage, for instance, of Spark Streaming[34] could have helped us in simplifying the parallelization of TWC and SRC, but still it would have not helped us in managing the interaction with our reference source and target databases, nor in guaranteeing the fault tolerance of the migration process, which does not depend only on the internal fault tolerance offered by Spark Streaming, but also on the way data are extracted and stored in the databases (see Section 6). Finally, still we would have had the problem of finding the optimal configuration for the Spark cluster, considering the specific characteristics of the data migration problem under analysis.

In the literature there are some works that try to support Big Data frameworks configuration problem. For instance, in Li et al. (2014) authors describe the challenges in configuring MapReduce parameters in an optimal way, highlight the difficulty of properly fine tuning MapReduce jobs, and propose an algorithm, which is able to dynamically configure MapReduce parameters to improve performance. So far, authors have focused on key parameters that affect task execution time, but the approach does not address the parameters that affect the overall application execution time (e.g., the number of mappers and reducers). Another similar approach is presented in Kroß et al. (2015) where authors exploit Layered Queuing Networks to analyse MapReduce applications. However, scalability and accuracy of the approach have not been demonstrated yet.

[33] http://azure.microsoft.com/en-us/services/documentdb/

[34] http://spark.apache.org/streaming/

In conclusion, despite the availability of the above frameworks, designers and developers still have to cope with complex fine tuning, customization, and interfacing issues that make building efficient data intensive applications still a cumbersome activity.

7.3 Modeling and QoS Analysis/Simulation Tools

Nowadays, there are various efforts in place aiming at offering both design-level analysis tools and simulation tools to support the development of applications. The most notable frameworks in this context are Palladio (Becker et al. 2009) that is an IDE offering support to architectural modeling activities and integrating a number of analysis tools as plug-ins, and MARTE (MG 2009) that is a UML profile standardized by OMG and allowing designers to model non functional characteristics of software systems. The resulting specification can then be analyzed/simulated through specialized techniques (e.g., Layered Queing Networks or Control Theory as described below) that are integrated with MARTE through proper model transformations (Bernardi et al. 2016). However, these approaches address traditional web-based applications and do not offer, at this time, specific support to even simple DI systems.

Recently, Kroß et al. (2015) proposed an extension of the Palladio Component Model to cope with MapReduce applications, which are then analysed through LQNs. However, scalability and accuracy of the approach has not yet been demonstrated.

An ongoing effort is also proposed by the DICE project,[35] in which we are involved, that focuses on the extension of MARTE to cope with DI applications in order to support their analysis, optimization and automatic deployment (Casale et al. 2015), but this work is still in its initial stages. Other approaches focus on analysing and predicting the behaviour of the data storage components as described in the following.

Rolia et al. (2009) used Layered Queuing Network (LQN) for studying the performance of a SAP multi-tiered ERP system and its database. LQNs can usually model better than QNs low-level features such as finite connections and buffer pools (Ardagna et al. 2014). However, an in-depth knowledge of the building components is usually required and high accuracy can be achieved only for the software configuration used for model calibration.

Also control theoretic approaches show their limitations in this context as they usually adopt a linear model working around an operation point (Tanelli et al. 2011). The papers (Abdelzaher et al. 2008; Lightstone et al. 2007) show two attempts of applying control theory linear models to the evaluation and control of relational databases. Such linear models may not be accurate enough to describe a database with widely different query workloads and time-varying resource contentions, where a small workload increase will disproportionally degrade the performance when the system is subject to a borderline overload condition (Menascé and Gomaa 2000; Herodotou et al. 2011).

Statistical Machine Learning (ML) techniques have been also successfully applied to model database and DaaS (Hacigumus et al. 2013). One of the earliest work has been presented in Shivam et al. (2006, 2007), where a database performance model is actively learned by an engine from the interaction of different allocated resources, data, and workload. More recent work is being developed by Duggan and others (see for instance (Duggan et al. 2014)) and is focusing on predicting performance in the presence of concurrent queries,

[35] http://www.dice-h2020.eu/

considering an analytical (and therefore quite stable) workload. ML approaches usually offer higher accuracy than analytical models like QNs, but they lack generalization capabilities (Didona and Romano 2015) and, hence, cannot be adopted to support the architectural analyses of an application in its early design stage.

For NoSQL systems, to the best of our knowledge, the work in Stewart et al. (2013) is the only one that, by adopting QN models, estimates storage nodes performance with a good level of accuracy in the estimation.

7.4 Summary of Discussion

In conclusion, in the development of Hegira4Cloud we have discovered that there are many aspects to be considered when facing the problem of managing large quantities of data. In the literature we found various relevant technologies and integrated some of them. However, we could not find enough support to identify and assess problems and issues *before* going through complete and multiple iterations of development and experimentation. In this section we have provided an overview of the state of the art in three of the main areas that have been very challenging for us and we have shown that there is still a lot to do to actually enable professionals to build DI systems in a predictable way, within time and budget.

8 Conclusion

In this paper we reported our experience in designing and developing a DI application supporting data migration between column-family based NoSQL databases.

Due to the lack of early design analysis tools and clear SLA of the DaaS systems we have adopted in the experiments, we needed to rely on an experiment-based action research approach. We reported the problems we observed and the solution we developed by adopting, unfortunately, a time consuming development-testing-reengineering approach. We hope that this experience can be valuable for other researchers and developers working in DI application area.

As a future work, our initial goal is to build an automated testing environment able to support the identification of quantitative models relying on stochastic machine learning. Models will be used to predict DI components performance characteristics and will be integrated in a development framework able to simulate, explore, and optimize a DI application architecture at design time. We think that a lot of interesting and challenging work is ahead, especially if we want to consider more complex application logic compared to the one of Hegira4Cloud and exploit frameworks for batch analysis (e.g., Hadoop) and real-time data stream (e.g., Spark Streaming or Apache Flink) that are gaining momentum in the DI world.

In terms of the specific development of Hegira4Cloud, we need to perform extensive experiments to show that the approach we have adopted to enable fault tolerance works in all possible conditions. Moreover, we plan to evolve the concept behind Hegira4Cloud to support live synchronisation of different DaaS and NoSQL databases.

Acknowledgments The authors would like to thank Stefano Ceri, Alfonso Fuggetta and Damian Andrew Tamburri for their advices and for reviewing preliminary versions of this paper. This work has been supported by the European Commission grant no. FP7-ICT-2011-8- 318484 (MODAClouds), by the Windows Azure Research Pass 2013 and by various Amazon grants for supporting research activities.

References

Abdelzaher T, Diao Y, Hellerstein J, Lu C, Zhu X (2008) Introduction to control theory and its application to computing systems. In: Liu Z, Xia C (eds) Performance Modeling and Engineering. Springer, USA, pp 185–215

Abadi D (2012) Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. IEEE Computer, 45(2)

ArchiveTeam (2012) Twitter Stream. https://ia601605.us.archive.org/10/items/archiveteam-twitter-stream-2012-12/archiveteam-twitter-2012-12.tar

Ardagna D, Casale G, Ciavotta M, Pérez JF, Wang W (2014) Quality-of-service in cloud computing, Modeling techniques and their applications, Journal of Internet Services and Applications

Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53:50–58

Atzeni P, Bellomarini L, Bugiotti F, Celli F, Gianforme G (2012) A runtime approach to model-generic translation of schema and data, Inf. Syst

Baskerville R, Myers M (2004) Special issue on action research in information systems: Making is research relevant to practice—foreword. MIS Q 28(3):329–335

Becker S, Koziolek H, Reussner R (2009) The palladio component model for model-driven performance prediction. J Syst Softw 82(1):3–22

Bernardi S, Dranca L, Merseguer J (2016) A model-driven approach to survivability requirement assessment for critical systems. In: Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability

Brewer E (2012) CAP twelve years later: How the rules have changed. Computer 45:23–29

Casale G, Ardagna D, Artac M, Barbier F, Nitto ED, Henry A, Iuhasz G, Joubert C, Merseguer J, Munteanu VI, Pérez JF, Petcu D, Rossi M, Sheridan C, Spais I, Vladuic D (2015) Dice: Quality-driven development of data-intensive cloud applications. In: Proceedings of the 7th International Workshop on Modelling in Software Engineering (MiSE)

Ceri S, Widom J (1993) Managing semantic heterogeneity with production rules and persistent queues. In: Proceedings of the Nineteenth International Conference on Very Large Data Bases, pp 108–119

Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2006) Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI '06, pp 15–15 USENIX Association

Chauhan A (2012) How the size of an entity is caclulated in Windows Azure table storage?. http://goo.gl/ch9YXu

Chen CP, Zhang C-Y (2014) Data-intensive applications, challenges, techniques and technologies: A survey on big data. Inf Sci 275(0):314–347

Cluet S, Connor RCH, Hull R, Maier D, Matthes F, Suciu D (1998) Panel session: Metadata for database interoperation. In: Proceedings of the 6th International Workshop on Database Programming Languages, DBLP-6, (London, UK, UK). Springer, pp 35–37

Das et al (2012) All aboard the databus!: Linkedin's scalable consistent change data capture platform. In: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12, (New York, NY, USA), pp 18:1–18:14, ACM

Das S, Narasayya V, Li F, Syamala M (2013) Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. In: Proceedings of the VLDB Endowment, vol 7, p 12. Very Large Data Bases Endowment Inc.

Didona D, Romano P (2015) Hybrid machine learning/analytical models for performance prediction: A tutorial. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015, pp 341–344

Duggan J, Papaemmanouil O, Cetintemel U, Upfal E (2014) Contender: A resource modeling approach for concurrent query performance prediction. In: EDBT, pp 109–120

Ferry N, Solberg A, Jamshidi P, Osman R, Wang W, Seycek S, Gligor V, Sucasa R, Abhervé A (2015) MODAClouds evaluation report–Final versivon. Deliverable D8.5.2, Available from http://www.modaclouds.eu/wp-content/uploads/2012/09/MODAClouds_D3.7.2_MODACloudsEvaluationReportFinalVersion1.pdf [accessed 5 January 2017]

Godfrey R et al (2014) Information technology – advanced message queuing protocol (AMQP) v1.0 specification. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64955

Gorton I, Klein J (2015) Distribution, data, deployment: Software architecture convergence in big data systems. IEEE Softw 32(3):78–85

Hacigumus H, Chi Y, Wu W, Zhu S, Tatemura J, Naughton JF (2013) Predicting query execution time: Are optimizer cost models really unusable? In: Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013), ICDE '13, (Washington, DC, USA), pp 1081–1092 IEEE Computer Society

Harizopoulos S, Abadi DJ, Madden S, Stonebraker M (2008) OLTP Through the Looking Glass, and What We Found There. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, (New York, NY, USA), pp 981–992 ACM

Herodotou H, Dong F, Babu S (2011) No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In: SoCC, p 18

Hill Z, Li J, Mao M, Ruiz-alvarez A, Humphrey M (2010) Early observations on the performance of windows azure. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp 367–376 ACM

Hunt P, Konar M, Junqueira FP, Reed B (2010a) ZooKeeper: Wait-free Coordination for Internet-scale Systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, (Berkeley, CA, USA), pp 11–11 USENIX Association

Kambatla K, Kollias G, Kumar V, Grama A (2014) Trends in big data analytics. J Parallel Distrib Comput 74(7):2561–2573. Special Issue on Perspectives on Parallel and Distributed Processing

Kent W (1983) A simple guide to five normal forms in relational database theory. Commun ACM 26:120–125

Kroß J, Brunnert A, Krcmar H (2015) Modeling big data systems by extending the palladio component model. Softwaretechnik-Trends 3:35

Li M, Zeng L, Meng S, Tan J, Zhang L, Butt AR, Fuller N (2014) Mronline: Mapreduce online performance tuning. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14, pp 165–176

Lightstone S, Surendra M, Diao Y, Parekh SS, Hellerstein JL, Rose K, Storm AJ, Garcia-Arellano C (2007) Control theory: a foundational technique for self managing databases. In: ICDE Workshops, pp 395–403

Manyika J, Chui M, Brown B, Bughin J, Dobbs R, Roxburgh C, Byers AH (2012) Big data: The next frontier for innovation, competition, and productivity. McKinsey Global Institute

Marr B (2015) Big Data: 20 Mind-Boggling Facts Everyone Must Read. http://www.forbes.com/sites/bernardmarr/2015/09/30/big-data-20-mind-boggling-facts-everyone-must-read. [Forbes Online; accessed January 2017]

Menascé DA, Gomaa H (2000) A method for design and performance modeling of client/server systems. IEEE Trans Softw Eng 26(11):1066–1085

MG (2009) Uml profile for marte: Modeling and analysis of real-time embedded systems

NIST Big Data Interoperability Framework (2015) Volume 6, Reference Architecture. doi:10.6028/NIST.SP.1500-6 [accessed 14 January 2017]

O'Brien R (1998) An overview of the methodological approach of action research

Picioroaga F, Nechifor S (2014) Modelling Smart City Urban Safety Planner - Final prototype design. Deliverable D8.5.2, Available from http://www.modaclouds.eu/wp-content/uploads/2012/09/MODACloudsD8.5.2_SmartCityUrbanSafetyPlannerDesignFinalPrototypeDesign.pdf [accessed 5 January 2017]

Popescu A (2010) Nosql at codemash – an interesting nosql categorization. http://nosql.mypopescu.com/post/396337069/presentation-nosql-codemash-an-interesting-nosql

Rolia J, Casale G, Krishnamurthy D, Dawson S, Kraft S (2009) Predictive modelling of sap erp applications: Challenges and solutions. In: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '09, (ICST, Brussels, Belgium, Belgium), pp 9:1–9:9, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering)

Sadalage PJ, Fowler M (2012) NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional 1st ed.

Scavuzzo M (2013) Interoperable data migration between NoSQL columnar databases, Master's thesis Politecnico di Milano

Scavuzzo M, Di Nitto E, Ceri S (2014) Interoperable data migration between nosql columnar databases. In: 18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOC Workshops 2014, Ulm, Germany, September 1-2, 2014, pp 154–162

Scavuzzo M, Tamburri DA, Di Nitto E (2016) Providing Big Data Applications with Fault-Tolerant Data Migration Across Heterogeneous NoSQL Databases. In: Proceedings of the Second International Workshop on BIG Data Software Engineering, (Austin, TX, USA)

Scoffield B (2010) Nosql – death to relational databases(?). Presentation at the CodeMash conference in Sandusky (Ohio) 2010-01-14

Shivam P, Babu S, Chase J (2006) Active and accelerated learning of cost models for optimizing scientific applications. In: Proceedings of the 32Nd International Conference on Very Large Data Bases, VLDB '06, pp 535–546 VLDB Endowment

Shivam P, Demberel A, Gunda P, Irwin DE, Grit LE, Yumerefendi AR, Babu S, Chase JS (2007) Automated and on-demand provisioning of virtual machines for database applications. In: SIGMOD Conference, pp 1079–1081

Stewart C, Chakrabarti A, Griffith R (2013) Zoolander: Efficiently meeting very strict, low-latency slos. In: ICAC, pp 265–277

Stonebraker M, Cetintemel U (2005) One Size Fits All: An Idea Whose Time Has Come and Gone. In: Proceedings of the 21st International Conference on Data Engineering, ICDE '05, (Washington, DC, USA), pp 2–11 IEEE Computer Society

Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P (2007) The End of an Architectural Era: (It's Time for a Complete Rewrite). In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, pp 1150–1160 VLDB Endowment

Szyperski C, Petitclerc M, Barga R (2016) Three experts on big data engineering. IEEE Softw 33:68–72

Tanelli M, Ardagna D, Lovera M (2011) Identification of LPV state space models for autonomic web service systems. IEEE Trans Contr Sys Techn 19(1):93–103

Terry DB, Prabhakaran V, Kotla R, Balakrishnan M, Aguilera MK, Abu-Libdeh H (2013) Consistency-based service level agreements for cloud storage. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, (New York, NY, USA), pp 309–324 ACM

Wong et al (2009) Oracle streams: A high performance implementation for near real time asynchronous replication. In: Ioannidis YE, Lee DL, Ng RT (eds) ICDE. IEEE, pp 1363–1374