Chapter 7
Fault-Tolerant Off-line Data Migration: The Hegira4Clouds Approach

Elisabetta Di Nitto and Marco Scavuzzo

7.1 Introduction

The Cloud offers the potential to support high scalability of applications. An increase in the application workload is typically handled by triggering the replication of its components so as to increase the application computational capability offered to users. Moreover, an increase in the amount of data to be handled can be easily managed by exploiting scalable DBMSs supporting partitioning of data on different nodes. These are the so called NoSQL databases that have been specifically built to offer scalability, high availability of data and tolerance to network partitions [9].

Unfortunately, when looking more closely at how NoSQL databases work, one realizes that they represent a good solution for scalability, but they do not offer mechanisms to allow migration among data stored in NoSQLs from different vendors. More specifically, data migration is not a new problem per se. It is a well established topic in relational databases world; this is mainly due to the standardization occurred at the data model level (with DDL) and at query level (with DML and DQL). There exist several tools (see, e.g., [2, 4, 6, 7]) which allow to migrate data across relational databases and, thanks to SQL, it is possible to preserve queries, compliant to the standard, even after the migration. On the contrary, in the NoSQL database field there exist no standard neither for interfaces nor for the data models and, as such, to the best of our knowledge, there are no tools which allow to perform data migration across different NoSQLs. Some databases provide tools to extract data from them (e.g., Google Bulkloader [3]), 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.

E. Di Nitto (✉) · M. Scavuzzo
Politecnico di Milano - DEIB, Piazza L. da Vinci 32, 20133 Milano, Italy
e-mail: elisabetta.dinitto@polimi.it

M. Scavuzzo
e-mail: marco.scavuzzo@polimi.it

© The Author(s) 2017

E. Di Nitto et al. (eds.), *Model-Driven Development and Operation of Multi-Cloud Applications*, PoliMI SpringerBriefs,
DOI 10.1007/978-3-319-46031-4_7

With our approach, that we call Hegira4Clouds,[1] we aim at providing a solution to the data migration problem in the context of NoSQL databases, trying to preserve, at the same time, the specific properties characterizing each NoSQL database. For the moment, we focus on column-family databases as they are among the most interesting class of NoSQL for their high level of scalability. Hegira4Clouds migration approach is based on the idea of extracting data from the source database, transforming them into an intermediate format, and, finally, translate and store them into the target database. Data transfer is fault tolerant as it enables the correct termination of the migration even in the presence of a failure within the migration infrastructure.

In the following of this chapter, we briefly present Hegira4Clouds intermediate format (Sect. 7.2) and its architecture, focusing, in particular, on the fault tolerance features (Sect. 7.3). Finally, we evaluate the approach (Sect. 7.4) and discuss conclusions and future work (Sect. 7.5).

7.2 Hegira4Clouds Intermediate Meta-Model

The Hegira4Clouds intermediate format is defined by an *intermediate meta-model* described in detail in [13]. It takes into account the features of the most widely used NoSQL and we have shown that it is sufficiently general for dealing with the features of so-called columnar and key-value NoSQL databases [8, 15]. Thanks to its definition, the adoption of a new NoSQL system in Hegira4Clouds requires only the development of the translator from this new NoSQL into the intermediate format and vice versa. Furthermore, thanks to this intermediate meta-model, Hegira4Clouds is able to preserve the data types, read consistency policies, and secondary indexes supported by the source database.

In particular, we preserve data types by keeping track of the type of each migrated data explicitly, even though that type is not available in the destination database. This is accomplished by performing the following procedure: data converted into the intermediate format are always serialized into a *property value field* and the original data type is stored as a string into a *property type field*. When data are converted from the intermediate format into the target one, if the destination database supports that particular data type, the value is deserialized. Otherwise, the value is kept serialized and it is up to the application level to correctly interpret (deserialize) the value according to the type field.

As extensively detailed in [10, 13], read consistency policies are handled through the concept of *Partition Group* (Fig. 7.1). Entities that require strong consistency on read operations will be assigned, in the intermediate format, to the same Partition Group value. Entities managed according to an eventual consistency policy will be assigned to different Partition Group values. When entities share the same Partition Group, if the target database supports strongly consistent read operations,

[1]Repository: https://github.com/deib-polimi/hegira-components/.

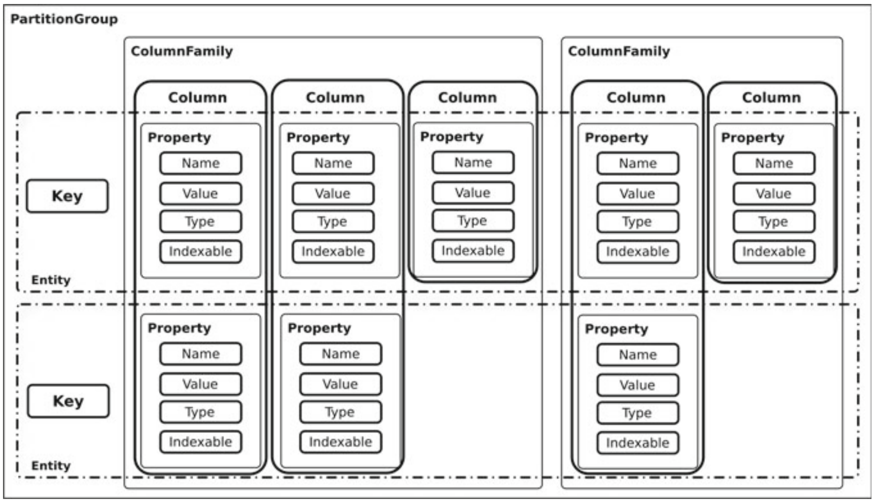

Fig. 7.1 Intermediate meta-model

then Hegira4Clouds adapts data accordingly (depending on the target database data-model). Otherwise, Hegira4Clouds simply persists the data so as that they will be read in an eventual consistent way, and creates an auxiliary data structure to preserve the consistency information.

Finally, secondary indexes are preserved across different database 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 into the target format, if the target database supports secondary indexes, the property is mapped consequently according to the specific interfaces provided by the target database. Otherwise, Hegira4Clouds creates an auxiliary data structure on the target database 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.

7.3 Architecture and Fault Tolerance Features

The Hegira4Clouds architecture is shown in Fig. 7.2. To provide scalablity and reliability, each component is decoupled from the other, and the interacting components communicate by means of a distributed queue. A *Source Reading Component (SRC)* extracts data from the source database, one entity at a time or in batch (if the source database supports batch operations) translates data into the intermediate format, by means of the respective *direct translator*, and puts the data in the *Metamodel queue*.

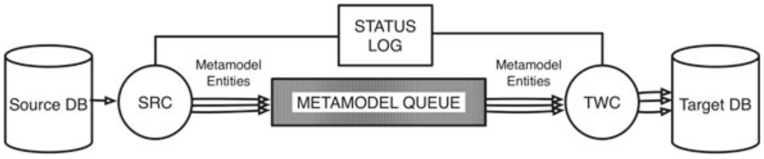

Fig. 7.2 Hegira4Clouds data migration architecture

This queue temporarily stores the data produced by the SRC so that other components can consume them at their own peace, thus allowing the system to cope with the different throughputs of the source and target databases. In parallel, a *Target Writing Component (TWC)* consumes the data from the queue and converts them into the target database data-model, thanks to an *inverse translator* (specific for each supported database). After conversion the data is stored in the target database. Hence the role of translators is that of mapping data back and forth between the source/target database and the intermediate format, performing the (de)serializations, checking for data types support, properly mapping indexes and adapting the data to preserve different read consistency policies. Two examples of translators (Google Datastore and Azure Tables) are extensively described in [10, 13]. SRC and TWC are organized in threads called *Source Reading Threads (SRT)* and *Target Writing Threads (TWT)*, respectively to achieve the maximum possible throughput.

Hegira4Clouds fault tolerance focuses on tolerating both databases reading/writing errors and outages (i.e., external faults) as well as crashes in the components of the migration system (i.e., internal faults).

Queue faults may be prevented by adopting a distributed, disk-persisted, queuing mechanism, so by assuming that this queue is able to automatically recover from faults of its replicas (that is the reason why we a adopt RabbitMQ, widely used in production environments).

Writing errors on the target database are addressed by the Metamodel queue; in particular, TWTs synchronously write data on the target database and send acknowledgment messages to the queue if the data were persisted correctly; only at this point, acknowledged data are removed from the queue. Thus, if an error occurs on the target database, another TWT (or a new TWC) can take over the specific write operations.

Reacting to reading errors in presence of faults on the source database, instead, is more difficult because of the heterogeneity of the different NoSQL databases; while some databases guarantee an absolute pointer to the data even after an error or a crash, thus enabling the possibility to restart the migration from the exact point in which it has been interrupted, some others (e.g., Google Datastore) do not.

Our approach to avoid restarting the migration from scratch consists in virtually partitioning the data in the source database, so that partitions containing a certain amount of data to be migrated can be retrieved in an ordered and unambiguous way, independently from the source NoSQL database that is being used. In this way, if there is an unrecoverable database error (i.e.,m external fault) or if the SRC crashes (i.e., internal fault), the migration can start from the last retrieved partition. This approach has been initially presented in [11] and is presented and evaluated in detail in the rest of this chapter. Of course, such an approach implies that data are stored in the databases according to a custom design. For this reason, Hegira4Clouds also supports a design-agnostic approach (see [14]) that is compatible with any kind of data design, but it is not able to react to unrecoverable source database faults or SRC faults (i.e., an internal fault).

7.3.1 Virtual Data Partitioning

Since the source database may not support absolute pointers to the data, in order to keep track of data that is being migrated, there must be some sort of shared knowledge between the application and Hegira4Clouds. For this reason, we define the concept of Virtual Data Partition (VDP), which is a logical grouping of entities contained in the source database. By making the assumption that the applications, using the source database, insert entities according to a sequential incremented (primary) key, it is possible to track the point where a data migration task was interrupted. In fact, by applying this technique, and storing only the last generated sequential id (*lastSeqNr*), it possible to unequivocally create VDPs and associate stored entities with them; in fact, by using an approach similar to paged virtual memory (virtual memory management) for operating systems, it is possible to map an ordered set of entities to a VDP (i.e., Eq. 7.2) and viceversa (i.e., Eq. 7.3).

To determine, at migration-time, the exact number of VDPs based on the last generated sequence number (*lastSeqNr*) and the user-defined partition size (*PS*) we use Eq. 7.1.

$$\#partitions = \left\lceil lastSeqNr/PS \right\rceil \tag{7.1}$$

$$VDPid_k = \left\lfloor key_{x,k}/PS \right\rfloor \tag{7.2}$$

We use Eq. 7.2 to calculate the id of the VDP containing the given entity (identified by its key, i.e., $key_{x,k}$). Finally, Eq. 7.3 can be used to calculate the first and last entity keys belonging to a given VDP (i.e., $VDPid_k$). Notice that:

$$\begin{aligned} key_{1,k} &= VDPid_k \times PS \\ &\vdots \\ key_{n,k} &= [(VDPid_k + 1) \times PS] - 1 \end{aligned} \tag{7.3}$$

- Since entities are inserted into the source database according to a sequential incremented key (generated in order to guarantee the global total order of the id), the entities contained in each VDP are ordered.

- The number of the VDPs is not fixed a priori, but it grows together with the inserted entities, and it is a factor of the number of inserted entities (*lastSeqNr*) and the maximum number of entities VDPs can contain (i.e., *PS*).
- The size of the VDPs, in terms of contained entities (and thus the number of VDPs, Eq. 7.1), can be determined at migration-time (by fixing a value for *PS*) and it can change from one migration to another, without affecting stored data.

Hence, for migrating data according to this approach, it suffices to read the last generated sequential number from a fault-tolerant, distributed storage, i.e., the *status log*, and decide the VDPs proper size; once done so, for each VDP, the SRC extracts the entities, from the source database, and executes the migration task.

If the source database supports range scan queries (e.g., HBase, Cassandra) it is possible, for each database specific *translator*, to request all the range of entities contained in a VDP, for example $VDPid_2$, with a single query, just by specifying its first (i.e., $key_{1,2}$) and last (i.e., $key_{n,2}$) entity keys. Otherwise, if the source database does not support range queries, the specific database *translator* requests each entity, contained in the VDP, one by one. In the first case, entities retrieval from the source database, is much faster than in the second case, because only a request, towards the database, is issued; while, in the second case, exactly *PS* requests are sent to the source database.

The limit of the VDP approach is that VDPs might also contain the ids of previously erased entities; while on the first hand, in case of range scans, this does not affect the performance of the migration task, since the source database handles missing entities in a range; on the other hand, if the source database does not support range scans, and the SRC has to issue a request per each entity contained in the VDP, the source database will return an error when trying to retrieve a previously deleted entity. The SRC skips the deleted entities that generate an error, but issuing queries also for deleted entities slows down the migration task. In the worst case, i.e., when all of the entities in a VDP have been erased, there may be a severe drop on the data migration overall throughput.

In order to keep track of the *migration status* (i.e., the number of entities correctly migrated towards the target database) and to allow for data synchronization (discussed in [12]), Hegira4Clouds exploits the VDPs. In particular, when the SRC is instructed to begin a migration task, it creates a *snapshot* of the source database, which is stored in the status log. A snapshot consists of: (a) a list of all the VDPs at the time the migration task was started (which depends on the value of *PS*, selected when the migration command was issued); (b) the status of each VDP, which can be of four types: “not migrated”, “under migration”, “migrated” and “synch”; (c) the last sequence number issued at the time the migration task was started.

When creating the snapshot, every VDP status is set to “not migrated”. Once the SRC starts to extract the entities relative to a given VDP, it sets the status of that VDP to “under migration”. When a TWT determines it has processed all entities relative to a given VDP, it sets that particular VDP status to “migrated”. The “synch” VDP status is used when a partition is being synchronized, but this is out of the scope of this chapter. A TWT is able to determine if a VDP has completely been processed by counting the effective number of processed entities for that VDP and comparing it with the number of entities the VDP actually contains (piggybacked on each metamodel entity and specific to different VDPs). Hence, each time a TWT processes an entity relative to a given VDP, it increments an associated counter; if the counter reaches the value piggybacked in those metamodel entities, then the TWT changes the VDP status to "migrated". In this way all Hegira4Clouds components are aware of the migration status at any point in time, and can therefore take the appropriate decisions in case of faults (and also of data synchronization, as described in [12]). Additional details about the snapshot management are provided in [14].

7.3.2 Recovering from Faults

Hegira4Clouds recovery approach assumes that there exists an external orchestrator (e.g., Mesosphere DCOS [5]) that acts as follows:

1. it monitors the statuses of Hegira4Clouds components, i.e., the SRC and the TWC;
2. if it detects a fault on the SRC, it waits until the TWC finishes to process all the messages in the Metamodel queue and starts a new SRC;
3. if it detects a fault on the TWC, it stops the SRC from reading data from the source database and restarts both components.
4. Once the components have been restarted, the orchestrator calls Hegira4Clouds recovery API.

Hegira4Clouds components, upon receiving a recovery command, in order to avoid inconsistencies during data migration, empty the Metamodel queue. Then, each components act as follows:

- the SRC
  - downloads the migration status from the status log;
  - for those VDPs whose status is "under migration", the SRC changes it to "not migrate" (this prevents inconsistencies from happening);
  - finally, it starts to extract data from the source database starting from the first VDP whose status is "not migrated".
- the TWC, just waits for the Metamodel queue to be filled by the SRC.

7.4 Evaluation: Migrating Tweets

This section evaluates Hegira4Clouds using a large data set extracted from Twitter. In particular, we stored into GAE Datastore 10,693,800 publicly available tweets [1] and then we ran Hegira4Clouds to migrate them into Azure Tables. The purpose of the experiment is to check if Hegira4Clouds is able to perform the partitioned data migration with an acceptable overhead (w.r.t. to the standard data migration [14]) and without introducing errors directly due to the migration process.

Experimental setup As mentioned before, our data set was composed of 10,693,800 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, with an extra sequential identifier (according to the specifics reported in Sect. 7.3.1) and a variable number of properties (with different data types). On average, each tweet on GAE Datastore was 3.05 KB. The total entities size was 31.1 GB. We tested Hegira4Clouds in two different scenarios:

1. *Standalone environent*: all of the migration system components, including the queue (RabbitMQ 3.4.6) and the status log (Apache ZooKeeper 3.5.4), were deployed inside an Azure VM.
2. *Distributed environent*: two equally-sized VMs in the same virtual network, one hosting the SRC, the TWC and the web-server exposing the REST APIs, and the other equipped with the queue and the status log.

In both scenarios the VMs were configured as follows: Ubuntu Server 12.04, located in Microsoft WE data center, with 4 CPU cores and 7 GB RAM.

Scenario 1: Standalone environment This test migrated data described above and used 32 TWTs to write data in parallel on Azure Tables and 8 SRTs to read data in parallel from Google Datastore. The main measured system metrics were (a) the total migration time and consequently the migration throughput (measured in entities per second), (b) the time needed by the SRC to extract the entities from the source database, convert and put them in the queue, and (c) the overall CPU utilization relative to all Hegira4Clouds components. We performed three different runs and computed the average of each metric. Moreover, in order to evaluate how predictable each run was with this configuration, we also computed the standard deviation for each metric (Table 7.1).

Scenario 2: Distributed environment In this scenario the environment setup was composed by two equally-sized VM, one, hegira1, executing an instance of RabbitMQ and ZooKeeper, the other, hegira2, hosting the SRC and TWC components, as well as the web-server exposing the REST APIs. The migrated data and the configuration parameters were the same of the previous scenario, but, additionally, we distinguished the CPU usages of the two VMs (Table 7.2).

Table 7.1 Partitioned data migration with standalone environment

# Run	Mig. time (s)	Mig. throughput (ent/s)	Ext. time (s)	Ext. throughput (ent/s)	%CPU used
1	13470	793.90	13469	793.96	49.4
2	16882	633.44	16880	633.52	49.02
3	17486	611.56	15248	701.32	38.46
Averages	15946	670.63	15199	703.58	45.63
Std. dev.	2165.44	99.56	1706.03	80.54	6.21

Table 7.2 Partitioned data migration with distributed environment

# Run	Mig. time (s)	Mig. throughput (ent/s)	Ext. time (s)	Ext. throughput (ent/s)	%CPU used hegira 1	%CPU used hegira 2
1	12075	885.61	12073	885.76	11.1	26.99
2	12187	877.47	12183	877.76	13.05	25.7
3	13995	764.11	13993	764.22	10.06	25.11
Averages	12752.33	838.58	12749.67	838.75	11.40	25.93
Std. Dev	1077.64	67.92	1078.16	67.98	1.52	0.96

7.5 Discussion and Conclusion

From the analysis of results we can conclude that Hegira4Clouds is suitable to handle and process huge quantities of data with a very high throughput. Deploying Hegira4Clouds on a distributed environment grants higher throughput; in fact, in scenario 2, the average migration time was almost 1 hour less and consequently the migration throughput was almost 170 ent/s faster. Moreover, by looking at the standard deviations, we can conclude that distributing Hegira4Clouds components has the benefit of providing more predictable migration performance. In fact, while in the first scenario we observe an average standard deviation corresponding almost to the 15 %, in the second scenario the standard deviation is almost halved to the 8 %.

Finally, by comparing the results obtained in Scenario 2 with those of the standard (i.e., non-partitioned) data migration [14] we can assert that the performance are almost the same and the adoption of the virtual data partitioning mechanism (together with the usage of a status log, i.e., ZooKeeper) has no tangible overhead on Hegira4Clouds.

The work on Hegira4Clouds is now focusing on how to manage synchronization between database replicas and on how to support data migration while the application using such data is continuing its normal execution.

References

1. ArchiveTeam (2012) Twitter Stream https://ia601605.us.archive.org/10/items/archiveteam-twitter-stream-2012-12/archiveteam-twitter-2012-12.tar
2. Flyway https://github.com/flyway/flyway
3. Google Bulkloader, https://chromium.googlesource.com/external/googleappengine/python/+/200fcb767bdc358a3acb5cf7cad1376fe69f12c5/google/appengine/tools/bulkloader.py
4. LiquiBase, http://www.liquibase.org
5. Mesosphere, https://mesosphere.com/
6. Mysql workbench: Database migration, http://www.mysql.it/products/workbench/migrate/
7. Oracle SQL Developer Migration, http://www.oracle.com/technetwork/database/migration/index-084442.html

8. Popescu A (2010, 02) Nosql at codemash—an interesting nosql categorization. http://nosql.mypopescu.com/post/396337069/presentation-nosql-codemash-an-interesting-nosql
9. Sadalage PJ, Fowler M (2012) NoSQL Distilled: a brief guide to the emerging world of polyglot persistence. Addison-Wesley Professional
10. Scavuzzo M (2013) Interoperable data migration between NoSQL columnar databases. Master's thesis, Politecnico di Milano
11. Scavuzzo M, Di Nitto E, Ardagna D Experiences and challenges in building a data intensive system for data migration
12. Scavuzzo M, Di Nitto E, Dominiak J (2015) Data synchronisation layer. MODAClouds deliverable D6.7, April 2015. http://www.modaclouds.eu/wp-content/uploads/2012/09/MODAClouds_D6.7_DataSynchronizationLayer.pdf
13. Scavuzzo M, Nitto ED, Ceri S (2014) Interoperable data migration between nosql columnar databases. In: Grossmann G, Hallé S, Karastoyanova D, Reichert M, Rinderle-Ma S (eds) 18th IEEE international enterprise distributed object computing conference workshops and demonstrations, EDOC Workshops 2014, Ulm, Germany, 1–2 Sep 2014. IEEE, pp. 154–162. http://dx.doi.org/10.1109/EDOCW.2014.32
14. 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 2nd international workshop on BIG Data Software Engineering (BIGDSE '16). ACM, New York, NY, USA, pp 26–32
15. Scoffield B (2014) Nosql—death to relational databases(?), January 2010, presentation at the CodeMash conference in Sandusky (Ohio), 14 Jan 2014. http://www.slideshare.net/bscofield/nosql-codemash-2010

Open Access This chapter is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, duplication, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the work's Creative Commons license, unless indicated otherwise in the credit line; if such material is not included in the work's Creative Commons license and the respective action is not permitted by statutory regulation, users will need to obtain permission from the license holder to duplicate, adapt or reproduce the material.