Executive Summary

This deliverable presents an interoperable migration system for columnar NoSQL databases, based on an intermediate Metamodel, capable of preserving both strong and weak consistency between data updates, secondary indexes and various datatypes. The adopted approach allows developers to easily add support for new databases. More specifically, this deliverable presents a description of the approach and describes the currently available prototypical implementation. The extension of the work to the more general case of data synchronization will be presented in Deliverable D6.7 that will be released at M30 of the project.
Members of the MODAClouds consortium:

Politecnico di Milano 
Stiftelsen Sintef 
Institutul E-Austria Timisoara 
Imperial College of Science, Technology and Medicine 
SOFTEAM 
Siemens Program and System Engineering 
BOC Information Systems GMBH 
Flexiant Limited 
ATOS Spain S.A. 
CA Technologies Development Spain S.A. 

Published MODAClouds documents

These documents are all available from the project website located at [http://www.modaclouds.eu/](http://www.modaclouds.eu/)
Contents

1 Introduction 4
   1.1 Context and objectives 4
   1.2 Structure of the document 5

2 Achievements 6

3 Conclusion 9
   3.1 Future works 9

Appendices 12

A Interoperable data migration between NoSQL columnar databases 13

B Advances in the interoperable migration system for NoSQL databases approach 22
Chapter 1

Introduction

1.1 Context and objectives

This document analyzes the current status of data synchronization techniques adopted in MODAClouds project. As stated in Deliverable D6.1 [1], Section 2.4: “Application data management and migration”, migration of data can occur either during the normal operations of the database (online data migration) or it can happen when it is out of service (offline data migration). In this document we illustrate the migration techniques currently adopted to perform offline data migration (of the whole dataset) for column-oriented NoSQL databases, a specific class of NoSQL databases described in Deliverable D4.1 [2]. We present a description of the approach and of the currently available prototypical implementation. The extension of the work to the more general case of data synchronization will be presented in Deliverable D6.7 that will be released at M30 of the project.

In general, migrating data from a database to the other not only implies the transfer of data from a database to the other, but it also requires an adaptation of data to the types, structures and data model offered by the target database, without impacting on data correctness and integrity properties. When talking about NoSQL data migration, since a common standard query language and data model do not exist, one should also take into account the possibility to adapt source code, in order to make it compatible with the new data format and database technology. Furthermore, each database provides schema properties – e.g. secondary indexes, consistency policies, etc. – that should be preserved whenever possible.

Typically, there are two general approaches when migrating data:

- Direct mapping, i.e. data from the source database are directly translated into the data structure of the destination database.
- Intermediate mapping, i.e. data are first translated into an intermediate format and then, from this, into the final one.

Even though the second approach requires an additional transformation (which is limited to the expressiveness of the intermediate model), it permits the easy support of a larger number of databases, since the number of translators that need to be implemented grows linearly with the number of supported databases. In contrast, the first approach requires a quadratic number of translators to be developed, but the expressiveness of the transformations can be customized.

The main goals of this deliverable is to illustrate the design and implementation of an extensible system capable of migrating data from any column-based NoSQL database to any other columnar NoSQL database. In order to do so, we will:

- Define an original Metamodel able to abstract each NoSQL columnar database data model characteristics and properties, preserving at the same time different consistency levels, secondary indexes and different data types, while migrating data among them.
• Provide a set of translators (as a proof-of-concept), which make use of the above Metamodel, that allows the migration of data between Google App Engine Datastore and Microsoft Windows Azure Tables.

1.2 Structure of the document

This paper is organized as follows:

• Chapter 2 synthesizes the design and implementation of both the Metamodel and the migration system.

• Chapter 3 contains the conclusion and the future planned work.

• The Appendices provide full details about the Metamodel and the migration system, as well as the performance tests that have been conducted.
Chapter 2

Achievements

By analyzing the characteristics of a large number of columnar NoSQL databases it is evident that their data model derives from the one proposed in the BigTable paper [3]. So, in order to perform a complete mapping between different columnar databases, the intermediate Metamodel should be based on BigTable data model too. Furthermore, some columnar databases introduced several properties which make the respective database more performant, or which guarantee different levels of consistency. Hence, the main properties we wish to maintain during migration are different data types and consistency policies and secondary indexes (for databases that support them). In the case in which the source database supports secondary indexes, whilst the target database does not, auxiliary data structures, preserving indexed properties, are automatically created in the destination database. The choice on how to design these data structures depends on the destination database. Further details about its design are provided in [4], also included in Appendix A.

Based on the definition of the previous Metamodel we have built a framework that supports the execution of direct translators, which transform data from the source data model to the Metamodel representation, and of inverse translators that transform data from the Metamodel representation to the target datamodel. Hence, for each supported database, two translators (performing the actual mappings) need to be implemented. As a proof of concept we have already built translators for two NoSQL databases: Google App Engine Datastore and Microsoft Windows Azure Tables. The complete mappings for these two databases can be found in [5].

On top of this framework we have built the migration system which tries to address the following requirements:

- In order to perform an offline migration of data (from a source database to a target one) which is as general as possible, the migration system should be independent from the applications using the data. Hence, it must sit in between the source and the target database and should be agnostic with respect to how data is generated.

- It should communicate with the databases by means of their official libraries, for maintenance reasons.

- It should be able to handle the different speeds occurring between the source and target databases – i.e. typically read and write operations take different times, in distributed systems read operations are faster than write operations – hence, extracted data may need to be buffered before being sent to the target database.

Given the above requirements, in order to have an independent system always available, it should be offered as a service on the cloud and should expose some methods to perform the switch over. Furthermore, since it should be decoupled from remote application logic, credentials needed to access the respective databases should reside inside the system. A general overview of the system is given in Figure 2.1.
The **Representation and Facade Layers** provide an entry point to the system – by means of REST API – and an interface to operate with different databases, in a general way. Furthermore, they contain the interfaces needed to request database switch over and which allow the extension of the system, adding support for new databases, and, in fact, create an abstraction layer that permits to seamlessly operate with any database.

The **Migration System Core** groups the business logic used to perform the actual migration between databases. In particular, it contains the framework (with Metamodel and translators) and implements a consumer-producer approach, as a mean for extracting data from a source database, converting them to the Metamodel representation and then storing them into the target database. Furthermore, this component is in charge of retrieving and storing information in each database, by means of vendor-specific libraries.

Typically write operations are slower than reads. Hence, extracting, converting and writing data into the destination, in a single synchronous operation, may result in the occurrence of timeouts and loss of data. Furthermore, source and target databases may be proprietary, hence tweaking timeout parameters is not an option. The producer/consumer approach is meant to address this issue through the use of a queue. As an added value, the introduction of a queue that provides fail-over mechanisms and redelivery policies permits to achieve better consistency over migrated data, in case of system faults or outages.

The producer extracts data from the source database, which is then passed to a direct translator, that transforms them into the Metamodel representation, and then saves them inside the queue. The consumer, upon noticing that elements are present in the queue, pulls and passes them to an inverse translator, which is in charge of converting them into destination database format. Finally, the resulting data are saved in the destination database.

In particular, we have decided to use the Single-Consumer-Multiple-Consumer (SCMP) Task Queue pattern, which allows to:

- execute the above write request in parallel to the target database, thus optimizing the migration system throughput (w.r.t. the single-producer-single-consumer approach, described in [A] and whose performance test results are also shown in [A]);
- mitigate the difference between read and write operations speeds.
Adding support for a new database simply involves the extension of an interface and the implementation of the proper translators to and from the Metamodel, as detailed in [5]; this will not only allow to migrate data to the new database, but it will also permit to preserve secondary indexes, data types and consistency policies.

Such a migration system can be deployed in two different scenarios:

1. as a standalone software, self-deployed on a IaaS environment and whose use is limited to just an user;
2. as a service (SaaS), running in a IaaS environment, but it can be used by several users.

The current implementation of the migration system addresses the first scenario; as a future work, we plan to extend the system so as to support more different users, operating at the same time, thus the migration system will be deployed as a SaaS. This last scenario should take into account security issues related to the storage of users’ credentials inside the migration system.

The general architecture of the system is described in Appendix A. Appendix B introduces the SCMP Task Queue. It has the objective of increasing the decoupling between the origin and the destination database and increasing the performance of the whole system.

<table>
<thead>
<tr>
<th>Objectives</th>
<th>Achievements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition and implementation of a Metamodel able to preserve each columnar database characteristic properties</td>
<td>Fully described in Appendix A</td>
</tr>
<tr>
<td>Implementation of a modular migration system, which makes use of the Metamodel previously defined, to abstract from each database specific implementation and perform actual offline data migration</td>
<td>The details about the migration system architecture are provided in Appendix A. Whereas, the changes adopted to introduce the SCMP Task Queue pattern are provided in Appendix B. Latest test analyzing the migration system performance can be found in Appendix B as well.</td>
</tr>
</tbody>
</table>
Chapter 3

Conclusion

This deliverable presents an approach that enables the migration of data among different NoSQL databases, focusing in particular on columnar NoSQL databases. As reported in Deliverable D4.1, there exists a wide number of different NoSQL databases solutions, for which a standard representation does not exist. Furthermore, each of these databases offers different properties and operations on data. Hence, data migration approaches should take these differences into account and try to provide a solution which is as interoperable as possible.

Our solution proposes an extensible migration system, composed of an intermediate Metamodel, that enables simple data migration among columnar NoSQL databases and which is able to preserve key characteristics, like consistency policies, secondary indexes and different data types. The actual version of the migration system provides translators for two databases: Google App Engine Datastore and Azure Table. Notwithstanding this, the premises are in place for additional support of other databases.

The migration system provides modular and extensible interfaces to the developers, which enable fast translators development and integration. Finally, the migration system exposes a set of REST API which allows external applications to interact with it.

Performance tests conducted in order to evaluate system performance have demonstrated that the overheads introduced by the migration system, in terms of latency and CPU usage, are acceptable (see B). Moreover, these tests have led to the conclusion that, as a future work, a more performant mechanism to serialize data should be studied and implemented in order to further improve the overall performance.

3.1 Future works

Short terms future works are focused on the improvement of the current offline migration system, hence they should investigate on the three main directives, briefly described in the following:

Migration system generalization In order to support more databases, new mappings need to be added to the migration system. This will also allow to perform parallel data migration over multiple destination databases.

Migration system functionalities extension Currently, the migration system supports only full data migration, hence, as a future step, we add the possibility to perform partial data migration, based on filters provided by the end user.

Migration system performance optimization As stated in Attachment B, the premises are in place for obtaining a significant gain in throughput towards target database, by implementing a more performant serialization solution.
In the long term we plan to extend the migration system by allowing to perform live migration, i.e. applications may continue to update data while the migration system is processing and migrating data. The main problem introduced by this feature, that will need to be addressed, is that of data consistency: updates on data during live migration may cause data inconsistency. Hence, some mechanism to perform data consistency check across different distributed databases should be studied.
Bibliography


Appendices
Appendix A

Interoperable data migration between NoSQL columnar databases
Interoperable data migration between NoSQL columnar databases

Marco Scavuzzo, Elisabetta Di Nitto and Stefano Ceri
Politecnico di Milano
DEIB
Piazza Leonardo da Vinci, 32, 20133 Milano, Italy
Email: marco.scavuzzo@mail.polimi.it, {elisabetta.dinitto, stefano.ceri}@polimi.it

Abstract—NoSQL databases have emerged as the solution to handle large quantities of user-generated contents still guaranteeing fault tolerance, availability and scalability. Each NoSQL database offers differentiated properties and characteristics as well as different data models and architectures. As a result, the development of applications exploiting such kind of technology is strictly dependent on the specific NoSQL solution being adopted, and the migration from a NoSQL to the other requires the development of ad-hoc code managing the transfer of data. In order to mitigate such issue, this paper proposes an interoperable migration system for columnar NoSQL databases. The proposed approach is based on an original Metamodel, capable of preserving both strong and weak consistency between data updates, secondary indexes and various data types. Moreover, the approach allows developers to easily add support for new databases.

Keywords—NoSQL databases; Data migration; Database as a Service (DaaS); Google App Engine Datastore; Azure Tables; BigTable; Amazon Dynamo; Amazon Dynamo DB

I. INTRODUCTION

With the advent of Web2.0 applications, in the last few years, the amount of user-generated content is growing exponentially and the applications that manage them require increasing levels of fault tolerance, availability and scalability to the underlying data management system. NoSQL databases have emerged to address these needs.

They differ from traditional relational databases in terms of their data model, typically quite simple, and in terms of their ability to scale by distributing data on different loosely coupled nodes.

Most relevant NoSQL databases combine approaches and solutions derived from the distributed systems theory (data partitioning and replication, failure detection and recovery, etc.), to provide the required properties and characteristics. Each of them proposes a different data model and architecture. Hence, based on the type of problem that needs to be solved, a certain database may be more suitable than another. At the moment of writing, the number of NoSQL databases is more than 150 [1], and there is not yet a commonly accepted categorization, even if several attempts have been made [2]–[6]. This is mainly due to the lack of standards for such kinds of databases.

Hence, when selecting the target database, developers should have clear ideas on the application requirements and on the specific characteristics of each possible candidate, thus, making the right choice in the first place may be difficult. Furthermore, new requisites or opportunities may arise during application development. All these may lead to the need for switching to a different database, but, given the significant differences between the various implementations, this may be problematic in terms of costs and time. This problem is commonly referred to as vendor lock-in.

In order to mitigate vendor lock-in, this paper proposes an interoperable migration system for NoSQL databases. In particular, we focus on a specific class of NoSQL, that is, columnar databases [6] and define: i) an original intermediate Metamodel that allows us to represent data in a common NoSQL format, ii) an extensible migration system which makes actual migrations possible and iii) a set of database-specific translators to enable the translation from a source database to the target one, passing through a representation in the intermediate Metamodel. The proposed Metamodel and translators are designed in such a way to preserve different levels of consistency, secondary indexes and different data types, during migration. The migration system provides a set of interfaces to easily add migration support to new databases. Moreover, it provides REST APIs which make it integrable with other applications. We have evaluated the overhead and performance of our migration system with two well known NoSQLs, Google App Engine Datastore, Microsoft Windows Azure Tables, showing that the approach preserves both strong and eventual consistency policies.

Consistently, the paper is organized as follows: Section II provides motivations and requirements for our work and an overview of the literature in the field. Section III is the core of the paper and presents the approach providing details on our intermediate Metamodel and on the architecture of the migration system. Section IV presents the specific translators we have developed so far. Finally, Section V presents an evaluation of our work and Section VI concludes the paper.

II. CONTEXT AND RELATED WORK

This section provides the necessary background and motivations for our NoSQL data migration system.
A. Motivations

Not only NoSQLs are different between each other in terms of technology, but also they are offered as a service (DaaS) by different cloud providers with different pricing policies. Both these issues cause vendor lock-in and increase the need to move from one provider/technology to the other.

Other typical reasons for migration are the need for a different data model, consistency policy or querying language, as well as costs reduction, better service level agreements and data redundancy on different vendor databases.

Our approach aims at fulfilling these needs by offering a system capable of performing data migration among different NoSQL solutions, preserving, whenever possible, fundamental characteristics like different levels of consistency or secondary indexes. This system lets database users change the service/technology to store data after an initial wrong choice, or in the event that a new vendor enters the market offering more adequate solutions.

B. Background and requirements

Data migration is the process of transferring data between two or more different databases. Typically, this task is performed automatically and requires only a limited interaction with the user. In general, migration of data can occur either during the normal operation of the database or it can happen when it is out of service. In this paper we concentrate exclusively on the second type of migration that we call data switch over.

Migrating data between NoSQL does not imply only their transfer from a database to the other, but it also requires an adaptation of data to the types, structures and data model offered by the target database, without impacting on data correctness and integrity properties.

We want to design a system capable of migrating data from any columnar database to any other columnar database. Columnar NoSQLs are those databases whose data model is similar to that of Google BigTable [7]. In general, their data model permits to achieve both horizontal and vertical scalability at the same time. This characteristic, among others, makes them particularly suited to “BigData” problems, living on clusters up to thousands of nodes.

This system should be independent from the applications using the data. Hence, it must sit in between the source and the target database and should be agnostic with respect to how data is generated. Furthermore, it should communicate with the databases by means of their official libraries, for maintenance and update reasons. Finally, it should be operated in a SaaS (Software as a Service) way, that is, it should be always available to serve migration needs and it should be able to handle the different latencies occurring between the source and target databases.

C. Related work

Data migration is a well-established area in traditional relational databases [8]. Vice versa, in the NoSQL context the situation is quite immature. Some DaaS vendors provide tools to import and export data from their database (see for instance the Google App Engine (GAE) Bulkloader 1). However, the usage of such tools is limited to the specific NoSQL for which they have been built and therefore does not address the vendor lock-in issue we want to tackle.

Other relevant approaches take a different perspective and tend to solve the problem of heterogeneity by providing a query mediation system or a wrapper generation tool; hence, they are suitable only if they are adopted while the applications exploiting NoSQLs are developed. An example of such approaches is the one by Atzeni et al., which proposes a common programming interface [9], based on a metamodel previously defined in paper [10], to be used as a library in projects that aim to use different NoSQL solutions. Another example is CPIM [11] a library that, by exploiting the Java Persistence API 2, provides an abstraction layer to operate with different Cloud services, such as NoSQL DaaS.

In contrast to these, our approach aims to be completely independent from the way applications are programmed and is meant to work exclusively at the level of data, by translating them from the source format into the intermediate Metamodel one and then into the target one. This is also what Torlone et al. do in the context of web data described in XML [12].

III. OUR APPROACH: AN INTEROPERABLE MIGRATION SYSTEM

In this section we propose our approach to interoperably migrate data between NoSQL columnar databases. Typically, there are two general approaches when trying to migrate data:

- direct mapping, that is, data from the source database are translated into the data structures of the target database;
- intermediate mapping, that is, data are translated into an intermediate format and then from this into the final one.

Even though the second approach requires an extra-transformation, it is more flexible once the structure of the intermediate format is properly defined. In fact, it can flexibly support a large number of translations, under the assumption that, for each supported database, two translators are written, one that transforms data from the database to the intermediate model and vice versa.

Thus, if \( N \) is the number of databases supported by the system, while with the second approach we would need \( 2 \cdot N \) mappings (i.e. the number of translators grows linearly),

1https://developers.google.com/appengine/docs/go/tools/uploadingdata
2http://en.wikipedia.org/wiki/Java_Persistence_API
with the direct mapping approach \( N \cdot (N - 1) \) translators would be needed (i.e. the number of translators grows quadratically).

Based on these considerations, we have decided to build our migration system by adopting an intermediate mapping approach. Thus, we have identified the Metamodel defining the intermediate format by studying the characteristics of a large number of existing databases and we have developed the framework that supports the execution of direct translators, that transform data from the source data model to the Metamodel, and of inverse translators, that transform data from the Metamodel to the target data model. The migration system is extensible as it offers simple programming mechanisms to interact with new databases and to develop new direct and inverse translators.

A. The Metamodel

By analyzing various types of columnar databases it is evident that their data model derives from the one proposed in the BigTable paper [7]. Thus, our intermediate Metamodel is based on the BigTable data model and it is enriched to take into account specific aspects of some columnar databases, most notably, different consistency policies, different data types and secondary indexes. To the best of our knowledge, the Metamodel presented in this paper is original, since there is no other system that allows migration between NoSQL databases and that preserves strong consistency as well as secondary indexes.

All things considered, the Metamodel in Figure 1 has been designed. A detailed description of each Metamodel component is given in the following subsections.

![Metamodel conceptual design](image)

1) **Property**: The basic unit for storing datum characteristics are the Properties. A property contains the single value, characterizing the datum, to be persisted along with its name; furthermore, it provides an explicit way to declare the datum value type and if that property should be indexable or not.

   The **Property name** field should contain the printable property name.

   The **Property type** field should contain a printable data type that will be used to deserialize the data contained inside the Property value field, if the destination database supports its original data type. Otherwise the data will be stored as an array of bytes (typically supported by any database).

   The **Property indexable** field contains a boolean value which states if data contained in the Property value field should be set as indexable or not inside the destination database (in case it supports indexes).

2) **Entity and Key**: An Entity is a grouping of Properties referring to the same element (i.e. the Entity). Each entity can contain an arbitrary number of properties.

   **Entity Keys** are used to distinguish Entities and index them; hence, they must be unique.

3) **Column**: A column is a grouping of similar properties belonging to different entities.

4) **Column Family**: As stated in the Google BigTable paper, Column Families are just groupings of different related Columns. Moreover, Column Families are disjoint, i.e. a Column may belong to just one Column Family at a time. In some databases, Column Families are used to guarantee the locality for data stored in it. The Metamodel provides a way to declare such groupings with a construct named after BigTable Column Family.

   Since one of the main characteristics of NoSQL databases is that of allowing to store sparse data, – i.e. data scheme is not fixed – Column Families may contain diverse number of Columns for every Entity.

5) **Partition Group**: Some databases provide strong consistency by simply letting users model data in constructs specific for each database. For instance, Google Datastore uses ancestor paths, whereas Azure Tables uses a combination of Partition Key and Table Name.

   For this reason, the Metamodel provides the Partition Group. Entities inside the same Partition Group are guaranteed to be stored in the destination database, in such a way that strong consistency (if supported) will be applicable to them, on every operation performed by the destination database.

B. System architecture

A general overview of the system is given in Figure 2. Given the requirements expressed in Section II, in order to have an independent system always available, it should be offered as a service on the cloud and should expose some methods to perform the switch over. Furthermore, since it should be decoupled from remote application logic, credentials needed to access the respective databases should reside inside the system.

The **Representation and Facade Layers** provide an entry point to the system – by means of REST API – and an interface to operate with different databases, in a general way. Furthermore, they contain the interfaces needed to request database switch over.
Figure 2. Migration System overview

The Migration System Core groups the business logic used to perform the actual migration between databases. The logic consists of two threads, a producer and a consumer, the intermediate Metamodel and two translators for each supported database. Furthermore, this component retrieves and stores information in each database, by means of vendor-specific libraries.

Typically write operations are slower than reads. Hence, extracting, converting and writing data into the destination, in a single synchronous operation, may result in the occurrence of timeouts and loss of data. The producer/consumer approach is meant to address this issue through the use of a queue. The producer extracts data from the source database, which is then passed to a direct translator, that transforms them into the Metamodel representation, and then saves them inside the queue. The consumer, upon noticing that elements are present in the queue, pulls and passes them to an inverse translator, which is in charge of converting them into destination database format. Finally, the resulting data are saved in the destination database.

Adding support for a new database simply involves the extension of an interface and the implementation of the proper translators to and from the Metamodel.

IV. DETAILS ON TRANSLATIONS

This section focuses on the transformation process to and from two well-known DaaS: GAE Datastore and Azure Tables. We first present the main differences between the data models offered by these two databases and then we describe, by means of an example, the translation algorithms implemented by the corresponding translators.

A. GAE Datastore & Azure Tables

In order to comply with the space limitations, the details of GAE Datastore and Azure Tables data models are not covered in this paper \(^3\). Nevertheless, we highlight the main differences between the two:

- **Support for data types.** GAE Datastore supports multi-valued data types – i.e. lists, arrays, Google custom data types, etc. – as well as simple ones; whereas, Azure Tables supports only single-valued data types.
- **Timestamps.** Azure Tables uses an accessible timestamp property which records when the respective entity was last modified. Even though GAE Datastore uses an internal timestamp too, in order to perform data-versioning, this is not accessible by the programmer.
- **Strong consistency.** GAE Datastore uses ancestor paths to guarantee strong consistency. An ancestor path is a hierarchy containing the keys and kinds of the entities which are parents of the given one. Hence, those entities are said to be in the same entity group. Azure Tables uses a combination of two elements to preserve strong consistency: Table name and Partition Key – i.e. Entities in the same table, with the same Partition Key are modified in a strongly consistent way.
- **Support for secondary indexes.** GAE Datastore allows the database designer to define these indexes on all data types except Binary and Blob and supports the execution of queries over indexed properties only. Conversely, Azure Tables does not permit to declare indexed properties, but it allows to express queries over any property, except on binary ones.

B. A Translation Example

To describe how the approach works, we consider an example of data migration (hence of translation) from Azure Tables to GAE Datastore. The definition of the complete translation algorithms, as well as some more examples to/from GAE Datastore and Amazon DynamoDB (a key-value database), can be found in [15].

Consider Table I, containing some data from the MiC application, we describe in Section V-A, and structured according to Azure Tables data model. The first three properties of each entity (Row Key, Partition Key and Timestamp) are mandatory. In particular, we can observe that the table contains two entities sharing the same Partition Key, which should be handled together according to the strong consistency semantics, and a third entity that is independent from the others, as it is associated to a different Partition Key.

The producer extracts data from Azure Tables, entity by entity, and passes them to Azure Table direct translator, which converts them into the Metamodel representation, as showed in Table II. Each Azure Tables entity is mapped to a Metamodel entity and the same thing happens with its Properties. In particular, Property mappings consist of the following steps:

- **Property data serialization:** since it is not possible to know in advance if source data type will be supported by the destination database, data are serialized and

\(^3\)A deep insight can be found on [13] and [14] respectively.
### Table I

**AZURE TABLES STARTING TABLE**

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Partition Key</th>
<th>Timestamp</th>
<th>email</th>
<th>topicName</th>
<th>ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:marco@polimi.it">marco@polimi.it</a></td>
<td>pk1</td>
<td>1392973421</td>
<td><a href="mailto:marco@polimi.it">marco@polimi.it</a></td>
<td>Politics</td>
<td>1,5,4,3,2</td>
</tr>
<tr>
<td><a href="mailto:giovanni@polimi.it">giovanni@polimi.it</a></td>
<td>pk2</td>
<td>1392985432</td>
<td><a href="mailto:giovanni@polimi.it">giovanni@polimi.it</a></td>
<td>Music</td>
<td>5,2,4,2,1</td>
</tr>
<tr>
<td><a href="mailto:paola@polimi.it">paola@polimi.it</a></td>
<td>pk1</td>
<td>1392996718</td>
<td><a href="mailto:paola@polimi.it">paola@polimi.it</a></td>
<td>Politics</td>
<td>3,1,1,4,2</td>
</tr>
</tbody>
</table>

### Table II

**METAMODEL DATA REPRESENTATION**

### Table III

**GAE DATASTORE GENERATED FICTITIOUS ENTITIES**

<table>
<thead>
<tr>
<th>Key</th>
<th>Ancestor Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>pk1</td>
<td>@UserRatings:pk1</td>
</tr>
<tr>
<td>pk2</td>
<td>@UserRatings:pk2</td>
</tr>
</tbody>
</table>

### Table IV

**GAE DATASTORE MIGRATED ENTITIES**

<table>
<thead>
<tr>
<th>Key</th>
<th>Ancestor Path</th>
<th>timestamp</th>
<th>email</th>
<th>topicName</th>
<th>ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:marco@polimi.it">marco@polimi.it</a> @UserRatings:pk1/UserRatings:<a href="mailto:marco@polimi.it">marco@polimi.it</a></td>
<td>1392973421</td>
<td><a href="mailto:marco@polimi.it">marco@polimi.it</a></td>
<td>Politics</td>
<td>1,5,4,3,2</td>
<td></td>
</tr>
<tr>
<td><a href="mailto:giovanni@polimi.it">giovanni@polimi.it</a> @UserRatings:pk2/UserRatings:<a href="mailto:giovanni@polimi.it">giovanni@polimi.it</a></td>
<td>1392985432</td>
<td><a href="mailto:giovanni@polimi.it">giovanni@polimi.it</a></td>
<td>Music</td>
<td>5,2,4,2,1</td>
<td></td>
</tr>
<tr>
<td><a href="mailto:paola@polimi.it">paola@polimi.it</a> @UserRatings:pk1/UserRatings:<a href="mailto:paola@polimi.it">paola@polimi.it</a></td>
<td>1392996718</td>
<td><a href="mailto:paola@polimi.it">paola@polimi.it</a></td>
<td>Politics</td>
<td>3,1,1,4,2</td>
<td></td>
</tr>
</tbody>
</table>
stored inside a Property value field. Notice that, in Table II, data has not been represented in their serialized form just for simplicity.

- **Original data type identification**: the type of the object previously serialized is identified and saved inside the respective type field.
- **Property indexing**: since Azure Tables does not provide secondary indexes, but it allows to express queries over every property, then the Property indexable field is set to true for all properties that are not of a binary type.

The Property is then associated to the corresponding Column and Column Family. Since Azure Tables does not offer this concept, all properties in the same Table are associated to the same Metamodel Column Family. Finally, the combination of entities Partition Keys and Table name, are mapped to the Metamodel Partition Group. In this example, two different Partition Groups are created.

Once a source entity is completely mapped to a Metamodel entity, it is stored in the shared queue and then extracted by a consumer, which requests a transformation to GAE Datastore format, to the respective inverse translator. This last step is showed in Tables III and IV.

When translating an entity from the Metamodel format to the Datastore format, a first issue is to handle the aspects concerning strong consistency. In particular, Entities in the same Metamodel Partition Group should be mapped to entities in the same ancestor path. Hence, when an entity belonging to a new Partition Group is considered, a fictitious root entity, should be created inside the Datastore in order to define a new ancestor path (see Table III in which two ancestor paths are created for our example). By doing so, all Metamodel entities, inside the same Partition Group, will be linked to the same root entity – i.e. they will be in the same ancestor path – once translated into GAE Datastore format (see Column “Ancestor Path” in Table IV).

The need for creating and handling the fictitious root entities results in the fact that, before writing a new entity in Datastore, one read operation, to check if the fictitious root exists, has to be performed and, if not, a preliminary write has to be executed to add it. This results in an increased overhead of writes, that in the worst case of independent Metamodel entities (i.e., when the eventual consistency semantic is selected) actually consists in one read and two write in the Datastore.

In general, when more than one Column Family is occurring, for each Column Family contained by a Metamodel Entity a new Datastore Entity, with the same Key but different Kind, is generated. In this example, a Metamodel Entity is mapped to just one Datastore Entity, since there is just one Column Family; and all of the generated entities will have the same Kind, as can be seen in Table IV.

The next step consists in mapping a Metamodel Entity Property to a Datastore Entity Property. Non-trivial aspects are the following. Before deserializing data contained by Column value field, a check on Datastore supported data types is performed.

1) If that particular data type is supported, then Property value field is deserialized and stored to the Datastore property value.
2) Otherwise, it is kept serialized and stored to the Datastore property value. Moreover, an auxiliary property, inside that Datastore Entity, is created and the original data type, contained in Metamodel Property type field, is inserted into it.

In this particular example, since Azure Tables data types are a subset of those supported by GAE Datastore, all data are deserialized and no new auxiliary property is created. Finally, since GAE Datastore supports secondary indexes, if a Property indexable field contains a true value (all Properties in our example), then the respective property on the Datastore should be set as indexable too.

V. Evaluation

During our research we have conducted tests aimed at verifying the generality of the adopted approach. In particular, we tested the behaviour of the migration system while migrating data from Azure Tables to GAE Datastore and viceversa, combining the different consistency policies. Furthermore, we have considered also the migration to and from Amazon DynamoDB that is a key-value database. These experiments are described in [15].

Here we focus on the evaluation of the overhead introduced by the system and of the total migration time. In order to do so, we consider the following parameters:

1) The throughput of data transiting in the system, i.e. entering entities, Metamodel entities in the Queue and exiting entities.
2) CPU usage concerning the execution of the migration platform.
3) Overall time needed for migration completion.
4) Time needed for getting data from source database, translating it to Metamodel representation and storing it in the queue – i.e. extraction and conversion time.

A. Case study: MiC weapp

In order to test our migration system, data stored by an application called “Meeting in the Cloud” (MiC)\(^4\) – a simple social networking web application – has been used. MiC is developed to be deployable both on GAE and Azure (for this purpose it exploits the CPIM library we have mentioned in Section II), but exploits the corresponding NoSQL DaaS in two different ways, as explained below.

MiC stores two different kinds of entities which contain strings, integer numbers and lists (of strings and numbers). When using GAE Datastore, MiC does not make use of ancestor paths. Hence data are stored so that future

\(^4\)GitHub repository - https://github.com/deib-polimi/mic-backend
operations will retrieve them in an eventually consistent way. Conversely, when using Azure Tables, MiC sets the same Partition Key for every entity stored inside the same table. This forces the database to store data so that future operations will retrieve them in a strongly consistent way.

B. Synthetic benchmarks

1) Experimental setup: The migration system has been deployed as a SaaS on Google Compute Engine. This is an IaaS platform and allows to create virtual machines with different characteristics. In order to carry out our tests, we have chosen a virtual machine with two virtual CPUs and 7.5GB of RAM, that runs a Linux-based operating system.

The toolset used for gathering measures during the operation of the migration system included the following components:

- The sysstat package has been adopted to measure the percentage of CPU used during tests.
- The log4j library has been integrated in the migration system to acquire information about the duration of the tests and the time needed to finish the production of the Metamodel objects to be stored in the queue.
- A customized library has been built to measure the size of objects stored in the queue.
- Google App Engine statistics on Datastore has been used to calculate the average dimension of stored entities and the total amount of data to be migrated. Unfortunately, we could not find a corresponding feature from the side of Azure Tables.

The virtual machine has been physically hosted by the Google datacenters in Western-Europe; whereas, Google does not provide any information about the physical location of the Datastore servers that host data. On the other hand, on Azure Tables, we decided to store our data in Western-Europe datacenters.

2) Evaluation scenarios: Since MiC uses different consistency policies on the two databases, we considered different scenarios for the evaluation. In this paper we present just two of them, the others can be found in [15].

1) Scenario 1: A migration from GAE Datastore to Azure Tables in order to ensure eventual consistency.
2) Scenario 2: A migration from Azure Tables to GAE Datastore in order to provide strong consistency.

SCENARIO 1: Google App Engine Datastore to Azure Tables migration

For the migration from GAE Datastore to Azure Tables three data sets of entities, with different size, have been chosen: 16MB, 64MB and 512MB.

The results are reported in Table V. From the results we notice that both the total migration time and the extraction and conversion time grow almost linearly with the amount of transferred entities; hence, the entities throughput is almost constant. Furthermore, the average usage of CPU is negligible (around 4-5% of the total CPU time).

<table>
<thead>
<tr>
<th>Source size</th>
<th>dataset #1</th>
<th>dataset #2</th>
<th>dataset #3</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Entities</td>
<td>1098</td>
<td>4270</td>
<td>34111</td>
</tr>
<tr>
<td>Migration time (sec)</td>
<td>33.643</td>
<td>34.604</td>
<td>34.653</td>
</tr>
<tr>
<td>Queued data (MB)</td>
<td>81.98</td>
<td>336.73</td>
<td>2792.81</td>
</tr>
<tr>
<td>Exiting data (MB)</td>
<td>69.00</td>
<td>283.79</td>
<td>2270.28</td>
</tr>
<tr>
<td>Avg. %CPU usage</td>
<td>4.74%</td>
<td>33.64%</td>
<td>4.111%</td>
</tr>
</tbody>
</table>

Table V

Tests were conducted starting from Monday at 9AM until Wednesday at 11PM.

SCENARIO 2: Azure Tables to Google App Engine Datastore migration preserving strong consistency

Table VI shows the results of the tests concerning the Tables to Datastore migration preserving strong consistency. Since Azure Tables does not give any information on the tables size, it is not possible to calculate an entity average size. Hence, an estimation, on the numbers of entities to be extracted, has been done, based on GAE Datastore entities.

With respect to the previous scenario we can observe that:

- The overall throughput for transferring all the entities, preserving strong consistency, is almost three times greater. This is due to the limitations imposed by the Datastore when operating with Entity Groups.
- Extraction and conversion times are almost identical.
- The average CPU utilization is lower than the previous scenario because of the lower Entities throughput.

Tests in this case are referring to a smaller number of entities if compared with the previous scenario. The reason is that, during tests with more 70000 entities, we experienced frequent unrecoverable crashes – i.e. HTTP 500 and 503 errors – not depending on the migration system, but due to the interaction with GAE Datastore that appeared to be unable to manage a large amount of HTTP requests within a given time interval. Hence, in this scenario we are migrating the maximum number of entities allowed by GAE Datastore within a single connection.

C. Considerations

In all considered performance scenarios, the time needed to perform a complete migration grows linearly with the amount of data to be transferred – i.e. source entities. By analyzing data stored inside the queue, we can assert that its average size is 4.43 times greater than the source data size. This is because of the metadata that are carried inside each Metamodel entity in order to preserve transactionality,
secondary indexes and data types, among the different databases.
CPU usage, during migration never exceeds 5%, hence the overhead on CPU is negligible.
Another key factor is the difference on read and write latencies between the two databases. In fact, the extraction and conversion time is less than the 0.1% of the time needed for the complete migration. This, in combination with Metamodel representation size, implies that queue size grows very quickly.

VI. CONCLUSION
In this paper, we presented an original approach that enables data migration among different NoSQL databases, focusing in particular on column-based ones.

The lack of standardizations for NoSQLs has brought to the proliferation of different database solutions, each with different characteristics and properties. In this context, preserving such characteristics, while migrating data, frequently causes vendor lock-in. Hence, we provide a solution to this problem, by proposing an extensible migration system, composed of an intermediate Metamodel and translators, that enables simple data migration among any columnar NoSQL database, and which is able to preserve key characteristics, like consistency policies, different data types and secondary indexes.
Tests conducted on the migration system aimed at evaluating system compatibility and performance. Performance tests have demonstrated that the overheads introduced by the migration system, in terms of latency and CPU usage, are negligible in every considered circumstance.
Future works should investigate on optimizing the current migration strategies and on continuous data migration, i.e. the possibility to perform data migration without switching off the applications using them.

ACKNOWLEDGMENT
The authors would like to thank Danilo Ardagna, Daniele Braga and Santo Lombardo for their support. The research reported in this article is partially supported by the European Commission grant no. FP7-ICT-2011-8-318484 (MODA-Clouds).

REFERENCES

<table>
<thead>
<tr>
<th>dataset #1</th>
<th>dataset #2</th>
<th>dataset #3</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Entities</td>
<td>9235</td>
<td>36940</td>
</tr>
<tr>
<td>Migration time (sec)</td>
<td>1402</td>
<td>5340</td>
</tr>
<tr>
<td>Entities throughput (ent/s)</td>
<td>6.587</td>
<td>6.918</td>
</tr>
<tr>
<td>Queued data (MB)</td>
<td>22.40</td>
<td>89.61</td>
</tr>
<tr>
<td>Extraction and Conversion time (sec)</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>Queued data throughput (KB/s)</td>
<td>2294.067</td>
<td>3058.627</td>
</tr>
<tr>
<td>Exiting data (MB)</td>
<td>2.92</td>
<td>3.07</td>
</tr>
<tr>
<td>Exiting data throughput (KB/s)</td>
<td>1.509</td>
<td>1.139</td>
</tr>
</tbody>
</table>

Tests started on a Thursday at 15PM and ended on the same day at about 10PM.

Table VI
MIGRATION FROM AZURE TABLES TO GAE DATASTORE PRESERVING STRONG CONSISTENCY

Tests started on a Thursday at 15PM and ended on the same day at about 10PM.

Table VI
MIGRATION FROM AZURE TABLES TO GAE DATASTORE PRESERVING STRONG CONSISTENCY

Tests started on a Thursday at 15PM and ended on the same day at about 10PM.

Table VI
MIGRATION FROM AZURE TABLES TO GAE DATASTORE PRESERVING STRONG CONSISTENCY

Tests started on a Thursday at 15PM and ended on the same day at about 10PM.
Appendix B

Advances in the interoperable migration system for NoSQL databases approach
Advances in the interoperable migration system for NoSQL databases approach

MARCO SCAVUZZO, ELISABETTA DI NITTO, DANilo ARDAGNA
Politecnico di Milano
marco.scavuzzo@mail.polimi.it, {elisabetta.dinitto, danilo.ardagna}@polimi.it

Abstract

Migrating data across NoSQL databases is a task which should take into account each database characteristic properties, as well as their inner architecture and performance. While some of these NoSQL databases are open-source and can be self-deployed, some others are not, and they are offered as a service (DaaS) by different Cloud providers. NoSQL DaaS architectures are proprietary, hence their functioning cannot be fully studied. This limits the possibilities to perform efficient data migration. In this appendix we propose an approach to seamlessly migrate data both in standard and DaaS NoSQLs. Furthermore, we perform several test on Azure Tables, a DaaS NoSQL database, in order to prove which are the maximum performance that can be achieved to migrate data towards this database.

I. INTRODUCTION

This appendix illustrates the work done to extend and optimize the interoperable migration system between NoSQL columnar databases described in [1].

The original migration system made use of a queue to overcome the different read and write latencies among source and destination database. Furthermore, a producer thread was in charge of extracting data from the source database, converting them into the common Metamodel representation and store them in the queue. A second thread (the consumer) would have extracted those data from the queue and, after having converted them to the destination database model, it would have persisted them in the target database. From the evaluation of the performance, described in [1], we thought about optimizing the whole migration time. In order to do so, since the extraction time was the 0.1% of the whole migration time, we decided to increase the number of consuming threads. Moreover, in order to provide better consistency over migrated data (especially in case of system faults or outages), we introduced a distributed message queue (RabbitMQ [2]) which provides fail-over mechanisms and redelivery policies.

The appendix is organized as follows: in Section II we present our approach providing details about the adoption of different consumer threads and the new queue. Section III presents the results obtained by testing the new migration system. In Section IV we discuss the results reported in the previous section. Finally Section V presents the conclusions and future planned works.

II. METHODS

As stated in [1], the migration system uses the producer-consumer approach as a mean for extracting data from a source database, converting them to a common representation (able to preserve databases specific characteristics: data types, consistency policies, secondary indexes) and then storing them to the target database.
In order to speed up the whole migration process, several consumer threads can be used to perform parallel data insertion into the target database. This way of proceeding generates several problems that should be taken into account:

- Data should be properly divided among threads;
- threads may contend the same data (race condition);
- data may be replicated into the destination database;
- in case of a system fault data should be properly migrated.

Since these problems are all addressed by message queues it has been decided to use a common open-source implementation of a distributed message queue called RabbitMQ [2]. In particular, a Single Consumer Multiple Producer (SCMP) Task Queue pattern has been adopted and a single RabbitMQ broker has been installed and configured together with the migration system.

The number of threads to be used for the migration can be chosen by the user by means of the migration system REST API. Each thread, upon initialization, opens a connection to the destination database and is inserted in a thread-poll waiting for a message to process. Messages are dispatched to threads according to the Round-Robin scheduling algorithm [3]; hence, on average, every consumer gets the same number of messages.

To prevent losses for messages already dispatched to consumers, especially if that consumer gets an error from the target database and cannot complete the insertion, an acknowledgement mechanism is used. An acknowledgement is sent back from the consumer to tell the migration system that a particular message has been received, processed and that the migration system is free to delete it from the queue. If a consumer dies without sending an ack, the migration system will understand that the message was not processed fully and will redeliver it to another consumer.

If the entire migration system crashes, together with RabbitMQ, messages stored in the queue may get lost. This can be prevented by the end user by means of a simple call to the migration system REST API. Since preventing data-loss upon system crash implies to persist data (stored in the queue) into the disk and not on RAM, the whole migration process is slowed down. Hence, the user, aware of this trade-off, can choose which fault-tolerance level should be adopted for that particular migration.

Producer and Consumers operations are almost unchanged with respect to the ones described in [1]; the only exception are the further respective serialization and deserialization operations that need to be done to save Metamodel objects inside RabbitMQ queue.

### III. Results

In order to evaluate the performance of the migration system, after having adopted the approaches described in Section II, several tests have been conducted.

In this section we report the conducted tests and, for each of them, we describe the test environment and present tests results. Discussions about the results are provided in Section IV.

#### I. Migration System performance tests

For these tests the migration system, together with RabbitMQ server, have been installed and configured inside a Windows Azure Virtual Machine (VM) whose characteristics are the following: Ubuntu Server 12.04, located in Microsoft Western Europe (WE) datacenter, with 4 CPU dedicated cores and 7 GB RAM.

The tests transferred data from Google App Engine (GAE) Datastore to Azure Tables. In
Table 1: Data Migration from GAE Datastore to Azure Tables

<table>
<thead>
<tr>
<th>Migration time (sec)</th>
<th>Exiting Entities throughput (ent/s)</th>
<th>Exiting data throughput (KB/s)</th>
<th>Extraction and Conversion time (sec)</th>
<th>Queued data throughput (KB/s)</th>
<th>Avg. %CPU usage</th>
<th>No. consumer threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>426</td>
<td>346.85</td>
<td>254.80</td>
<td>117</td>
<td>2947.11</td>
<td>24.10%</td>
<td>10</td>
</tr>
<tr>
<td>321</td>
<td>460.30</td>
<td>338.14</td>
<td>158</td>
<td>2182.35</td>
<td>33.06%</td>
<td>20</td>
</tr>
<tr>
<td>306</td>
<td>482.87</td>
<td>354.72</td>
<td>173</td>
<td>1993.13</td>
<td>35.56%</td>
<td>40</td>
</tr>
<tr>
<td>304</td>
<td>486.05</td>
<td>357.05</td>
<td>176</td>
<td>1959.16</td>
<td>36.23%</td>
<td>60</td>
</tr>
</tbody>
</table>

In particular, we migrated 147758 entities, corresponding to 64 MB\(^1\), originally persisted on GAE Datastore in an eventual consistent way. Exiting Azure Tables Entities size is 106 MB\(^2\); whereas data stored inside the queue (representing Metamodel entities) is 336.73 MB. Finally, messages stored in the queue, in these tests, are not persisted to disk, but are kept on RAM.

Tests results are reported in Table 1. Each test considers five aspects:
1. The number of consumer threads, writing in parallel in Azure Tables;
2. The throughput of data transiting in the system, i.e. Metamodel entities in the Queue and exiting entities.
3. CPU usage.
4. Overall time needed for migration completion.
5. Time needed for getting data from source database, translating it to Metamodel representation and storing it in the queue – i.e. extraction and conversion time.

For each number of consumer threads, three different runs of the same test have been conducted; results shown in Table 1 report the average of these three runs for each variable.

Several tools have been used to conduct these tests:
- `sysstat` package to measure the percentage of CPU used during tests.
- `log4j` library, integrated in the migration system, gives information about the duration of the tests and the time needed to finish the production of the Metamodel objects to be stored in the queue.
- A custom library\(^3\) to measure the size of objects stored in the queue and the size of Azure Tables entities\(^3\).
- GAE Datastore Statistics to calculate the average dimension of entities to be migrated.

II. Windows Azure VM network test

In order to find the bottlenecks, which prevent the migration system to achieve better results, several other tests have been conducted. In this subsection we consider tests aimed at measuring the maximum network bandwidth for a Windows Azure VM; whereas, in the following sub-section we tested the maximum number of parallel writes that is possible to make in the same Azure Tables account.

For these tests we deployed two VMs on Windows Azure cloud infrastructure with the following configuration: Ubuntu Server 12.04, located in the same Microsoft Western Europe (WE) datacenter (not in the same virtual network), with 4 CPU dedicated cores and 7 GB RAM. Furthermore, we deployed another VM instance on Google Compute Engine Cloud infrastructure with the

\(^1\)According to GAE Datastore Statistics
\(^2\)According to an article [4] appeared in the official Microsoft Blog
\(^3\)According to an article [4] appeared in the official Microsoft Blog
following characteristics: Debian 7, hosted in Google Western Europe datacenter, with 2 dedicated virtual CPUs and 7.5GB of RAM.

On each of these VM we installed a tool named iperf [5], able to measure the maximum achievable bandwidth on IP networks among two instances.

With a first test we measured the maximum incoming bandwidth from the Compute Engine VM to Windows Azure VM, trying to understand if this could be the bottleneck of the system. We found that the maximum incoming bandwidth in this case (on an average of five runs) was 418 Mb/s.

Table 2: Azure Tables entities throughput with queue

<table>
<thead>
<tr>
<th>Entity avg. size</th>
<th>Entities Total Size (MB)</th>
<th>No. consumer threads</th>
<th>Transfer time(s)</th>
<th>Throughput (ent/s)</th>
<th>Throughput (KB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>754 Byte</td>
<td>106</td>
<td>10</td>
<td>217.3</td>
<td>680.0</td>
<td>499.5</td>
</tr>
<tr>
<td>754 Byte</td>
<td>106</td>
<td>20</td>
<td>121.3</td>
<td>1218.1</td>
<td>894.8</td>
</tr>
<tr>
<td>754 Byte</td>
<td>106</td>
<td>40</td>
<td>102.3</td>
<td>1444.4</td>
<td>1061.0</td>
</tr>
<tr>
<td>754 Byte</td>
<td>106</td>
<td>60</td>
<td>122.0</td>
<td>1211.1</td>
<td>889.7</td>
</tr>
<tr>
<td>754 Byte</td>
<td>106</td>
<td>192</td>
<td>127.3</td>
<td>1160.7</td>
<td>852.7</td>
</tr>
<tr>
<td>1KB</td>
<td>152</td>
<td>10</td>
<td>230.0</td>
<td>642.4</td>
<td>676.7</td>
</tr>
<tr>
<td>1KB</td>
<td>152</td>
<td>20</td>
<td>156.0</td>
<td>947.2</td>
<td>997.7</td>
</tr>
<tr>
<td>1KB</td>
<td>152</td>
<td>40</td>
<td>129.0</td>
<td>1145.4</td>
<td>1206.6</td>
</tr>
<tr>
<td>4KB</td>
<td>580</td>
<td>10</td>
<td>244.0</td>
<td>605.6</td>
<td>2434.1</td>
</tr>
<tr>
<td>4KB</td>
<td>580</td>
<td>20</td>
<td>148.0</td>
<td>998.4</td>
<td>4013.0</td>
</tr>
<tr>
<td>4KB</td>
<td>580</td>
<td>40</td>
<td>139.0</td>
<td>1063.0</td>
<td>4272.8</td>
</tr>
<tr>
<td>4KB</td>
<td>580</td>
<td>60</td>
<td>120.0</td>
<td>1231.3</td>
<td>4949.3</td>
</tr>
<tr>
<td>4KB</td>
<td>580</td>
<td>192</td>
<td>116.6</td>
<td>1267.2</td>
<td>5093.7</td>
</tr>
</tbody>
</table>

With a second test we measured the maximum outgoing bandwidth from a Windows Azure VM to another, trying to understand if there could have been a limit on it. On an average of five runs, we found that this outgoing bandwidth is 712 Mb/s.

III. Azure Tables writes tests

Since the tests reported in previous sub-section proved that the incoming and outgoing network connections cannot be a bottleneck for the system, in this section we measure the maximum number of parallel writes that a VM is able to make in the same Azure Table account.

In particular we developed two tools able to write entities to Azure Table in parallel. Each tool differ in its implementation in that, one uses an internal queue to store entities to be written in parallel by different consumers; in the other tool there’s no queue, and the different threads are in charge of both production and consumption of the entities.

The configuration of the VM used for these tests is the same of the previous sub-sections tests, and the log4j library as well as the custom library to measure entities size have been used.

Tests results with the first tool (with queue) are reported in Table 2; whereas tests conducted with the second tool (without the queue) are shown in Table 3. Each of these transfers 147758 entities and considers four aspects:

1. The number of consumer threads, writing in parallel in Azure Tables;
2. Overall time needed for writing the entities in Azure Tables;
3. The maximum throughput needed for transferring those entities;
Table 3: Azure Tables entities throughput without queue

<table>
<thead>
<tr>
<th>Entity avg. size</th>
<th>Entities Total Size (MB)</th>
<th>No. consumer threads</th>
<th>Transfer time(s)</th>
<th>Throughput (ent/s)</th>
<th>Throughput (KB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>152</td>
<td>10</td>
<td>210.6</td>
<td>701.6</td>
<td>739.1</td>
</tr>
<tr>
<td>1KB</td>
<td>152</td>
<td>20</td>
<td>130.3</td>
<td>1134.0</td>
<td>1194.5</td>
</tr>
<tr>
<td>1KB</td>
<td>152</td>
<td>40</td>
<td>125.0</td>
<td>1182.1</td>
<td>1245.2</td>
</tr>
</tbody>
</table>

4. Entities average size as well as Entities total size.
For each test with the same number of threads, three different runs have been conducted; hence, results shown in Table 2 and 3 report the average of these three runs for each variable.

IV. Discussion

In this section we discuss the obtained test results reported in Section III.

About the introduction of SPMC Task Queue pattern
As reported in Section II in this appendix we show the improvements achieved with respect to the migration system presented in [1]. The main innovation introduced since that paper are the SPMC Task Queue pattern and the tests conducted to find the main bottlenecks.

In order to comply with space constraints, we could not insert in [1] the results of the migration tests from GAE Datastore to Azure Tables with Single Producer Single Consumer (SPSC) Task Queue pattern; notwithstanding this, those results are reported in thesis [6]. From the analysis of the results about the tests on the two patterns, we can notice that, with the adoption of the SPMC approach, the overall migration time is decreased up to 3.6 times at the cost of a greater average CPU utilization (it grows up to 7.6 times) and of an increased “Extraction and conversion time” (to values up to 5.6 times). Still, the system is underutilized, but we seem to have reached the maximum number of threads useful for maximizing the entities throughput.

Maximum achievable performance
The throughput tests reported in Section III aimed at measuring the maximum achievable performance when writing data to Azure Tables. Once established, we can identify the ideal migration system performance. From our tests, we did not find a significant difference between the tests that used the queue and the ones which did not. Furthermore, it seems that Azure Tables imposes a limit on the maximum number of write requests that it is possible to send to the database, independently from the transferred entities average size. In fact, it appears that it is possible to write up to 1444 entities per second, provided that the VM, where the migration system is installed, has enough network bandwidth (both in upload and download) available.
Network bandwidth tests proved that available network bandwidth is not a problem for a large-size VM deployed on Windows Azure cloud infrastructure.

Finding the bottleneck
From tests described in the previous section, we can assert that the maximum throughput achieved by the migration system is 2.5 times lower than the maximum achievable one. As stated previously, the network bandwidth, as well as Azure Tables write requests limitations, are not a bottleneck for the migration system.
By analyzing the tests conducted on RabbitMQ message queue [7] we can conclude that this queue can handle more messages than those we used and with a higher message rate (up to 10000 messages per second). Hence, the queue does not seem to limit the whole migration system performance.

Having excluded all of the above causes, we think the double serialization and deserialization performed by the migration system, in order to preserve databases characteristics (data types, secondary indexes and consistency policies) and to store data in the Task Queue, may cause a significant drop in performance. Future tests will investigate this possibility and will try to provide a better solution.

V. CONCLUSIONS

In this appendix we reported the significant performance improvements we achieved by introducing a Single-Producer-Multiple-Consumer Task Queue pattern inside the migration system. Other improvements introduced by the usage of a message queue included a better response to faults and fair message dispatching to consumers.

Several tests have been conducted in order to measure the new system performance and to find other possible improvements that it is possible implement. This study has led us to the conclusion that, as a future work, a more performant mechanism to serialize (and deserialize) Metamodel objects should be studied and implemented. Moreover, further tests measuring the migration from Azure Tables to GAE Datastore should be performed, in order to further improve and optimize the whole migration system.

REFERENCES


