Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Interoperable data migration between NoSQL columnar databases

Advisor: Stefano Ceri
Co-Advisor: Elisabetta Di Nitto

Master thesis by:
Marco Scavuzzo - matr. 782545

Academic Year 2012-2013

Dedico questa tesi ai miei genitori che hanno sempre assecondato le mie scelte e che non mi hanno mai fatto mancare il loro sostegno.

Ringraziamenti

Desidero innanzitutto ringraziare il Professore Ceri e la Professoressa Di Nitto per avermi concesso la possibilità di svolgere questo lavoro di tesi. Li ringrazio inoltre per la disponibilità e l'impegno con cui mi hanno seguito durante tutto il periodo di svolgimento del lavoro di tesi.
Ringrazio inoltre il Professore Ardagna per l'aiuto e la disponibilità fornitami per effettuare i test descritti in questo elaborato.
Ringrazio Santo Lombardo per i preziosi consigli e per il tempo concessomi per discutere degli argomenti trattati in questa tesi.
Voglio infine ringraziare tutti gli amici che mi sono stati accanto durante tutti questi anni, per il loro supporto e la loro amicizia incondizionata che, diverse volte, ha saputo distogliere la mia attenzione dagli ostacoli che si sono presentati.

Milano, 18 Dicembre 2013

Marco

Estratto

Con l'avvento delle applicazioni Web2.0, la quantità di contenuti generati dagli utenti sta crescendo esponenzialmente. Inoltre, il maggiore utilizzo di web services, in concomitanza con una migliore aspettativa di livelli di servizio, hanno portato all'adozione di approcci atti a garantire maggiore tolleranza ai guasti, availability e scalabilità. Dunque, per far fronte a moli enormi di dati e garantire, al contempo, i requisiti di cui sopra, si è cominciato a far uso di database NoSQL.

I più noti, tra questi, utilizzano tecniche e soluzioni proprie della teoria dei sistemi distribuiti, al fine di fornire caratteristiche e proprietà differenti. Infatti, ognuno di questi si contraddistingue per un data model ed una architettura differente. Dunque, in base al problema da risolvere, un certo database può risultare più adatto di un altro. Diversi tentativi per classificare i vari database NoSQL sono stati fatti, tuttavia non vi è ancora una categorizzazione comunemente accettata. Ciò è dovuto principalmente alla mancanza di uno standard per questi database. Dunque, all'atto dello sviluppo di una applicazione, si dovrebbe avere una chiara visione circa il panorama dei database NoSQL. Ma, dato il numero di essi, può essere difficile fare da subito una scelta corretta. Inoltre, nuovi requisiti potrebbero emergere durante la fase di sviluppo.

Quando gli effetti di una scelta errata del database si manifestano, potrebbe essere problematico, in termini di tempi e costi, effettuare un cambiamento. Questo problema è noto come vendor lock-in e, al fine di porvi rimedio, in questo lavoro di tesi si descrive un sistema interoperabile per la migrazione di dati tra database NoSQL. In particolare, ci si concentra sui database a colonne, e si definisce un metamodello originale che permette di migrare dati tra database di questo tipo, ed anche Key-Value.

Il metamodello proposto è progettato in modo da preservare diversi livelli di consistenza, indici secondari e tipi di dato. Inoltre, si descrive lo sviluppo di un sistema di migrazione estensibile, che permette agli sviluppatori di supportare nuovi database, senza che questi necessitino di conoscere il funzionamento degli altri.

Abstract

With the advent of Web2.0 applications, the amount of user generated content is growing exponentially. Moreover, the increasing number of people using web services, together with more stringent service level expectations, have led to the adoption of approaches able to grant fault tolerance, availability and scalability. Hence, in order to handle very huge amount of data and granting, at the same time, all of the above requirements, NoSQL databases have emerged.

The most relevant NoSQL databases combine approaches and solutions, relative to the distributed systems theory, to provide different properties and characteristics. Each of them proposes different data models and architectures. Hence, based on the type of problem that needs to be solved, a certain database may be more suitable than another. Several attempts have been made in order to categorize NoSQL databases, but there is not yet a commonly accepted categorization. This is mainly due to the lack of standards for NoSQL databases.

Hence, when developing an application, one should have a clear vision of all NoSQL solutions. Given the number of existing databases, it may be difficult to make the right choice in the first place. Furthermore, new requisites may arise during application development.

If the effects of a poor database choice become evident, switching to another database may be problematic in terms of costs and time. This problem is known as vendor lock-in and, in order to mitigate it, this thesis proposes an interoperable migration system for NoSQL databases. In particular, we focus on columnar NoSQL databases and define an orginal metamodel that allows to transfer data among this class of databases.

The proposed metamodel is designed in such a way to preserve different levels of consistency, secondary indexes and different data types, during migration. Moreover, we develop an extensible system which allows developers to easily add support for new databases, without requiring any knowledge about the other databases. For this reason, guidelines about the implementation of proper database translators are provided, together with examples of how to support Key-Value databases too.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of Web2.0 applications, in the last few years, the amount of user generated content is growing exponentially. Furthermore, the increasing number of people using web services, together with more stringent service level expectations, have led to the adoption of approaches able to grant fault tolerance, availability and scalability. Hence, in order to handle very huge amount of data and granting, at the same time, all of the above requirements, NoSQL databases have emerged.
The most relevant NoSQL databases combine approaches and solutions, relative to the distributed systems theory, to provide different properties and characteristics. Each of them proposes different data models and architectures. 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 the 150; several attempts have been made in order to categorize them, but there is not yet a commonly accepted categorization. This is mainly due to the lack of standards for NoSQL databases.
Hence, when developing a new application, that should make use of such databases, one should have a clear vision of all NoSQL solutions. Given the number of existing databases, it may be difficult to make the right choice in the first place. Furthermore, new requisites may arise during application development. If the effects of a poor database choice become evident, switching to another database 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 thesis proposes an interoperable migration system for NoSQL databases. In particular, we focus on columnar NoSQL databases and define an orginal metamodel that allows to transfer

data among this class of databases. In addition, the proposed metamodel is designed in such a way to preserve different levels of consistency, secondary indexes and different data types, during migration.
Furthermore, we develop an extensible system which allows developers to easily add support for new databases, without requiring any knowledge about the other databases. For this reason, guidelines about the implementation of proper database translators are provided, together with examples of how to support Key-Value databases too.
The usage of this migration system would let database user choose the proper service to store his data, even after an initial wrong choice, or in the event that a new vendor enters the market offering more adequate solutions.
Furthermore, data migration will provide more flexibility, letting users choose how to "move" their data, based on the type of result they want to achieve. For example, a more expressive query language may be needed and part of the application logic should be delegated to the database. Hence, data should be made compatible with, and moved to, a given database technology.
Or, maybe, big computational efforts need to be made in order to extrapolate knowledge from data. Hence, a database which natively supports MapReduce should be chosen and data should be properly migrated into it.
In general, this migration system will unleash the potentiality generated by all NoSQL database solutions, independently from the original data location.

Original Contributions

This work includes the following original contributions:

- An innovative metamodel able to abstract data model characteristics and properties of column-based NoSQL databases.
- Complete translators, which make use of the above metamodel, that allow to migrate data between Google App Engine Datastore, Microsoft Windows Azure Tables and Amazon DynamoDB (Key-Value).
  Furthermore, guidelines on how to write custom translators are provided.
- An extensible migration system which makes actual migrations possible, and which provides a set of interfaces to easily add support to new

databases. Moreover, it provides REST APIs which make it integrable with other applications.

Outline of the Thesis

This thesis is organized as follows:

- Chapter 2 describes the current state of the art about NoSQL databases. At first, we provide motivations that make users choose NoSQL databases over relational ones. Subsequently, we report the most accepted categorization for NoSQL databases and briefly discuss it.
  Then, we recognize two papers as the main directives that led to the development of the majority of NoSQL databases; hence, we briefly analyze them in order to highlight the most common problems that arise when designing such scalable and distributed databases.
  Finally, we discuss the main properties of the databases used as use cases in the whole thesis.

- Chapter 3 is dedicated to the design and the development of the migration system.
  At first, we define the main requirements the system should respond to. Then, the system underlying architecture is described. Hence, we report the original metamodel that enables data migration between different NoSQL columnar databases.
  Finally, a deep analysis of the system design process is conducted.

- In Chapter 4 we study how translators from a source database to the metamodel, and viceversa, should be developed in general. Furthermore, we discuss the design and the implementation of two real translators: Google App Engine Datastore and Azure Tables.

- In Chapter 5, we conduct several tests on the whole migration system.
  A first test takes a real application as use case and tries to migrate its data among two different databases. Then, we check whether the application still works.
  Furthermore, we conduct several tests both in private server and in IaaS cloud environments, in order to determine the real performance of the

migration system.
Subsequently, in light of previous tests, we make a theoretical comparison between database-to-database translators, and translators which make use of the Metamodel described in Chapter 3. Finally, we study approaches in order to extend the proposed Metamodel, in particular we show how a Key-Value database, with different consistency policies, can be supported.

- Chapter 6 draws the conclusions on the entire work and proposes several planned future works.

Chapter 2

State of the art

2.1 Introduction

This chapter starts by listing the typical approaches used to scale Web2.0 applications that make use of relational databases, highlighting the limitations and difficulties. Hence, it introduces NoSQL databases which permit to achieve better results with fewer efforts. After having categorized the different "families" of NoSQL databases, it then continues by motivating the reasons that led us to concentrate on columnar databases.
In Section 2.3 we analyze the two most important papers in this field, Google BigTable paper and Amazon Dynamo paper, that laid the foundations for the implementation of the most used NoSQL databases. Furthermore, databases that have been influenced by each of these papers are briefly discussed.
Finally, the last two sections describe the NoSQL databases used, in this thesis, to develop a middleware capable of migrating data between columnar databases in an interoperable way.

2.2 NoSQL Motivations

NoSQL databases have recently started to become popular due to the advent of Web 2.0 applications and the consequent need to manage a huge amount of data.
This kind of databases try to cope with some limitations of RDBMS that come up as the volume of data increases. In these cases aspects like: fault tolerance, availability across distributed data sources, scalability and consistency become really important. Besides, several usage patterns do not require com-

plex queries and management functionalities typically provided by RDBMS.
In the past years these problems where addressed incrementally; the very first approach was to try to scale vertically by adding computational power to the system, but still this solution suffers of the single point of failure issue. Sooner or later, the problem would arise again and, at that point, you tried to scale horizontally by adding clusters to the database. This generates data replication and consistency problems, among the different instances, that should be taken into account.
Furthermore, since RDBMS must comply with the ACID properties, distributed transactions need to be orchestrated across multiple nodes.
All of the previous strategies prefer consistency over availability. So, in order to respond to the increasing needs of scalability and availability a further step was taken: denormalizing the data inside the database violating the five normal forms adopted when designing the application.
Finally one of the liabilities, in some scenarios, for RDBMS is given from the constraints on the schema:

1. A fixed schema, in some cases, may not serve the application needs well, making it impossible to be modified at runtime.

2. A properly normalized schema imposes to create additional tables to express "many-to-many" relationships which implies the execution of join queries, which in turn slow down the whole system.

2.2.1 NoSQL common characteristics

When talking about NoSQL databases there are two schools of thought [13]. One saying that NoSQL stands for "Not SQL", the second one, instead, asserts that the word NoSQL means "Not Only SQL". And actually the second school have proven to be right because some NoSQL databases actually use SQL-like languages to express queries (for example Couchbase uses an SQL-like language called UnQL).
NoSQL databases are used to store and retrieve large amounts of data in a more efficient way, with respect to relational ones. This increase in performance is achieved by loosening consistency constraints, horizontal scaling, a finer control over availability and the lack of join queries.

ACID is a set of properties (Availability, Consistency, Isolation and Durability) which are guaranteed during operations over data inside a transaction. ACID rules, generally used by relational databases, do not apply anymore in NoSQL ones; instead NoSQL databases guarantee eventual consistency or, in some databases, transactions limited to single data items.
NoSQL databases must cope with the CAP or Brewer's Theorem which states that given the properties of consistency, availability and partition tolerance, a distributed computer system can satisfy up to two of those at the same time.

Consistency: a distributed system is said to be consistent if every node sees the same data at the same time.

Availability: means that the system is designed to keep working even if a failure to some nodes happens.

Partition tolerance: implies the ability of the system (divided in two or more partitions) to continue its work even if these partitions cannot communicate with each other.

For several number of applications and use-cases (large web-applications, e-commerce platforms, etc.) availability and partition tolerance are more important than consistency, hence BASE approaches are applied.
BASE is an acronym for: **B**asically **A**vailable, **S**oft-state and **E**ventual consistent; which means that "an application works basically all the time (basically available), does not have to be consistent all the time (soft-state) but will be in some known state eventually (eventual consistent)".

2.2.2 NoSQL classifications

In the last few years a lot of new NoSQL databases have been developed. Each of them tries to fit several requirements (e.g. scalability, maintenance and feature-set) differently. Some of these databases enhance Google BigTable, some others move from Amazon Dynamo or a combination of both. Also databases based on completely different technology (i.e. graphs theory) have emerged. Several classifications have been proposed, one [12] of the most generic is indicated in Table 2.1.

Table 2.1: Classifications – Categorization and Comparison

	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value Stores	high	high	high	none	variable (none)
Column stores	high	high	moderate	low	minimal
Document stores	high	variable (high)	high	low	variable (low)
Graph databases	variable	variable	high	high	graph theory
Relational databases	variable	variable	low	moderate	relational algebra

Key/Value stores Key/Value stores are similar to data-structures like Maps or Hashtables, since they permit to retrieve information (values) given their key.
The most simple implementations just store uninterpreted values – i.e. arrays of bytes – associated to a key and allow queries only by key. Whereas, more complex implementations try to interpret data to be stored as Values, hence they permit more complex queries over Values. Furthermore, some implementations permit scan queries over Keys.
Data can be stored in two different fashions, depending on the implementation that is being used: in-memory (e.g. Redis) or on-disk.

Column-oriented databases These databases owe their name to the data model proposed by Google BigTable paper. Data are stored inside structures named Columns, which in turn are contained inside Column Families, and are indexed by Key. Typically, data inside the same Column (for every Key) are persisted together.
Since this thesis will focus on columnar databases, a more in-depth description will be given in Paragraph 2.2.3 and also in Section 2.3.

Document-oriented databases Document databases are mainly used to store semi-structured data which are called Documents. Each document may contain several unique ID fields, used to index other fields that can be of any type, simple or nested, including other documents. Typical documents formats are: JSON, BSON, YAML or XML.

This kind of databases allows to express queries over fields contained in the document.
Each implementation differs from the others in that they may index data differently, or they may provide diverse kind of queries; but, typically, the main differences reside in consistency and replication mechanisms and on design details.

Graph databases Graph databases are based on graphs theory and are typically used to represent highly interconnected data. Information are stored inside graphs; a node is the entity under consideration and its information is stored inside the so called properties (key/value pairs). It is possible to express relationships between nodes by means of edges.
A field where graph databases are becoming popular is that of social networks, since ever-growing relationships among people are the main requisite, hence fast and scalable approaches, that just graph databases can provide, are needed.

By the end of 2012 the number of the NoSQL products was about 120 and obviously not all of them may perfectly match the categorization above.

2.2.3 Columnar NoSQL databases

This thesis will concentrate on Column stores for several reasons:

- They are typically released in two fashions: database as a service (DaaS) – e.g. Google Datastore, Microsoft Tables (Hybrid solutions, not completely columnar) – and standalone – e.g. Cassandra, Hbase, etc.
- Useful to store semi-structured data designed to scale to a very large size. In fact they are typically used for managing Big Data that can be partitioned both horizontally and vertically.
- Thanks to projection on columns, only the data really needed can be retrieved, maximizing the throughput.
- They are particularly useful in scenarios where individual row queries need to be performed and aggregation operations are not necessary. This is because of column families, i.e. items that need to be co-accessed can be put in the same column family.

Moreover, some columnar NoSQL databases guarantee strong consistency under particular conditions, but, in general, they handle data in an eventually consistent way.
Usually queries may filter on keys values or column name (projection), but some of these databases permit to declare secondary indexes in order to filter queries by means of an arbitrary column value.
Columnar oriented NoSQL databases are typically based on Google BigTable Paper, hence they share some characteristics, such as: Memtables and SSTables usage, a multi-dimensional map to index properties, column families – in order to group columns of the same type – and timestamp – to maintain data up-to-date and perform versioning.

2.3 Column-based NoSQL databases origins

This section describes two of the most important papers that have deeply influenced the development of the most recent NoSQL databases: Google BigTable and Amazon Dynamo
Some of the problems and considerations addressed by these papers are directly adopted by different NoSQL databases, some other instead have been solved differently. Anyhow, at the end of each paragraph, the most important databases that have been influenced by the respective paper are briefly discussed.

2.3.1 Google BigTable

Google BigTable has been categorized in different ways, in fact it is commonly referred to as "wide columnar store", or as an "extensible record store", or even as an "entity-value-attribute" (EAV) datastore. The description given by Google in its paper ([7]) about BigTable is: "a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers". Their main purpose was to "scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time".

Underlying technology

BigTable makes use of several technologies:

Google File System (GFS): a proprietary, distributed file system used to persist data and logs.

Cluster management system: used "for scheduling jobs, managing resources on shared machines, dealing with machine failures and monitoring machine status".

Memtable and SSTable file format: Memtables are data structures (write-back caches) that contain recently committed updates which should be kept in memory and retrieved, when necessary, by key. When a Memtable is full data is written to disk, in a Sorted String Table (SSTable), containing key-value pairs, that, once written, become immutable. The stored keys are of the type $(rowkey, columnkey, timestamp)$. After a certain amount of time, SSTables are compacted in order to limit the number of SSTables which have to be considered for read operations.

Chubby : a "highly-available and distributed lock service" used for synchronizing accesses to shared resources, elect a master and discover tablets in a BigTable cluster. Furthermore, it is used to store metadata that contains information like column-families for each table, access control lists (ACL), etc. A chubby cell is made of a set of servers, called replicas; each of which maintains copies of a single database. Among these servers, a master server is elected by means of the Paxsos algorithm (Consensus algorithm for multiple nodes to agree on a value despite failures or delays). When a client application sends read or write requests (using the chubby library) it is received by the master. If the request is a write request, it gets propagated to the replicas throughout Paxos algorithm, and after the write have reached the majority of replicas, the master replies to it. In case of a read request, the master responds immediately. If the master server dies a new one is elected among the other nodes, using

the Paxos algorithm.
Hence, Paxos is used by BigTable to provide reliability and availability.

Data model

BigTable data model can be described as "a sparse, distributed, persistent multidimensional sorted map"; data are stored in binary arrays and indexed by a triple $(rowkey, columnkey, timestamp)$.

Rows Rows are stored in lexicographic order by means of their row-key and are dynamically partitioned into tablets, which represent "the unit of distribution and load balancing" in BigTable. Tablets are stored in tablet servers which handle read and write requests and split tablets that have grown too large (100-200MB). It is up to the client application to exploit tablet property in order to speed up read operations. Client applications communicate with BigTable by means of a client library, which is in charge of look up tablet servers, and then operate on data that should be read or written.

Columns Columns grouped in the same column family can benefit of several properties:

- Since they "form the basic unit of access control", it is possible to set specific privileges in order to read, write and add column families.
- They typically store the same kind of data.
- BigTable compresses column family data together.
- Column families should be declared before data can be stored in it, and they should have a printable name.
- Paper authors suggest to keep the number of column family small.
- Locality groups are a special feature which permits to declare group of column-families that should be accessed together. This forces BigTable to create an SSTable for each locality group within a tablet. This process is used to speed up reads, reducing the number of disk seeks and the quantity of data to be retrieved.

Timestamp Timestamps are used to perform versioning over data; data are stored in decreasing order of their timestamp value so that the most recent version can be retrieved. An automatic garbage-collector is in charge of deleting the older revisions.

Operations The typical operations on data supported by BigTable are Create, Read, Update, Delete (CRUD) and scans by range on keys; but searches by column are not supported.

Consistency considerations

BigTables provides transactions on a single row basis: "Every read or write of data under a single row key is atomic (regardless of the number of columns being read or written in the row), a design decision that makes it easier for clients to reason about the system behaviour in the presence of concurrent updates to the same row". Instead, operations on multiple rows are eventually consistent.

BigTable influences

Google BigTable has influenced many other NoSQL databases, in particular for the data model design. Some common examples are: Cassandra, HBase, Hypertable.
HBase and Hypertable are BigTable open-source implementations which work practically as described in Google BigTable paper.

Cassandra, instead, employes several techniques borrowed both from BigTable and Amazon Dynamo. Cassandra data model is similar to the one described in BigTable paper, with the exception of super-columns which are a further data structure used to group similar columns inside the same column-family. Even though Cassandra does not use a distributed file system like GFS, it represents data in memory and on disk as BigTable does: firstly new datum is written to a persistent commit log, then to an in-memory data structure and, if it reaches a certain threshold of size, it gets persisted to disk. All writes on disk are sequential and indexed to achieve better performance, like BigTable does with block-indexes in SSTables.

2.3.2 Amazon Dynamo

Another important paper that contributed to the development of the latest NoSQL databases is Amazon Dynamo paper [8]. Even if it describes an actual database, used internally by Amazon applications, it is not meant to be used for external projects since it is tailored to Amazon specific needs and infrastructure architecture.
For example, the fact that it does not support any security mechanism, or that data versioning must be handled by the client applications, are just two of the factors that make it unusable for non-Amazon projects.
This is the reason why Amazon developed two other database systems, specifically address to developers using its cloud architecture: SimpleDB (deprecated) and DynamoDB.
Nonetheless, Amazon Dynamo paper describes some useful techniques widely adopted by other NoSQL databases, for example by Cassandra. As will be seen below, some Cassandra common problems (such as partitioning, failures detection and handling, data replication and versioning) have been solved taking inspiration from Amazon Dynamo paper.

Amazon Dynamo is a key-value storage used to store data, up to 1MB of size per pair, in binary form (uninterpreted byte array).
Since storage systems that provide ACID properties tend to have poor availability, Dynamo team has decided not to provide this kind of guarantees by operating with weaker consistency, lack of isolation and thus offering only single key update.
Dynamo operates in a non-hostile environment, hence no security mechanisms, such as authorization and authentication, have been implemented.
Dynamo replicates data in a peer-to-peer-like fashion by storing enough information at each node ($O(1)$ routing) in order to perform routing efficiently (Zero hop DHT).

Data partitioning

Partitioning is one of the problems for which the paper proposes a solution. In order to achieve incremental scalability, the system needs to dynamically

partition data across the nodes (i.e. storage hosts) that compose the system; hence a variant of consistent hashing technique is employed.

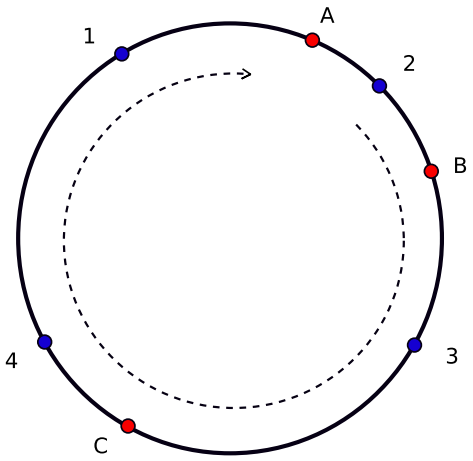

Figure 2.1: Consistent Hashing - initial situation

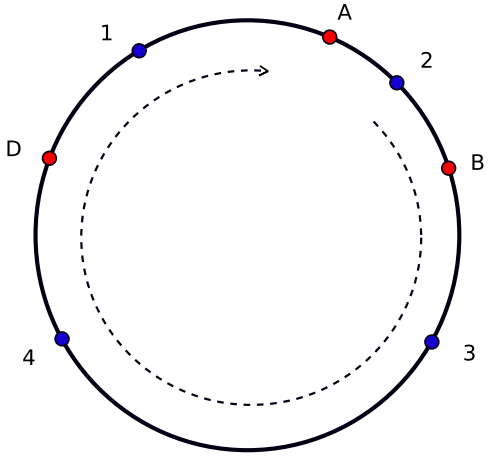

Figure 2.2: Consistent Hashing - situation after node join and departure

Consistent Hashing In consistent hashing the output range of a common hash function is treated like a ring.
Each node composing the system is placed randomly on this ring. Instead, items to be stored gets a position on the ring after having hashed their key value. Which item is mapped to which node is decided by moving clockwise on the ring; for example, in Figure 2.1 item 4 and 1 are mapped to node A, item 2 to node B and so on.
Operating in this way, when the number of nodes changes it is not necessary to remap all of the nodes, but just a subset of them (the ones contained by the neighbours). This is what happens in Figure 2.2: node C leaves the system and a new node called D enters, so now items 3 and 4 are mapped to node D and the other items remain unchanged.
The basic algorithm presents two issues:

1. non-uniform data load and distribution due to random position assignment for nodes in the ring.

2. node heterogeneity is not taken in consideration, i.e. nodes with different storage space or computational power are treated in the same way.

Virtual nodes In order to solve both of these problems, Dynamo uses a variant of consistent hashing algorithm which contemplates the concept of virtual nodes. When a node joins the system it gets assigned to multiple points in the ring, i.e. a virtual node.
The number of virtual nodes, for a physical node, can be defined individually based on its hardware configuration, thus it does not need to be the same for every node on the ring.
In case of a node departure, data stored in it becomes unavailable unless it has been replicated to other nodes before. The symmetric problem happens when a new node joins the system and a neighbour node is not responsible anymore for some of its data.
By introducing a replication factor (r) both of these problems can be solved, since items are replicated to the next r physical nodes in clockwise direction. Dynamo by default sets $r = 3$, so if we consider item 1, in Figure 2.1, it gets stored by node A and replicated on node B and C.
The list of nodes in charge of storing a particular item is called preference list and it typically contains just physical nodes.

Failure detection

Nodes joins and departures are noticed thanks to a gossip-based protocol which maintains an eventual consistent view of the nodes in the system. Thanks to the gossip protocol it is possible to avoid having a centralized registry for storing membership and node liveness information (like BigTable does).

Failure handling

Sloppy Quorum In order to keep the various replicas consistent, Dynamo uses a sloppy quorum system based on the proper configuration of two parameters: R and W, which respectively represent the minimum number of nodes that must participate in a successful read/write operation.
So, if N is the number of the first healthy nodes in the preference list, in order to have a proper configured quorum system, the two parameters should be set such that $R + W > N$.
If a write operation completes without errors the client can be sure that data

has been correctly replicated to, at lest, $W - 1$ nodes. But, if there is an error in the write operation, the client cannot be sure if the operation has not succeeded at all (i.e. none of the nodes processed the operation) or less than $W - 1$ processed it correctly. In the latter case, the nodes, which have received the write request correctly, are now in charge of replicating the data correctly to the other nodes and then agree on the most recent version.
Therefore, the only way for the client to achieve a consistent state is to re-issue the write operation.
The overall system latency is determined by the slowest node among those in the R and W group.

Hinted Handoff If a node becomes unreachable during a write operation data is stored by another node, usually not responsible for this data item, and gets hinted. When the proper node will be reachable again the item will be synchronized. Whereas the node, having received the update as a substitute, may then delete the hinted update from its local, separated, database. This strategy is called Hinted Handoff.

Failure recovering

In order to prevent threats to durability, which may cause replicated data to become inconsistent, Dynamo employs Merkle trees.
Merkle trees or hash trees consist of leaves which are hashes of the values of the individual keys and parents nodes which are hashes of their children. Merkle trees are typically used to efficiently verify the content of large data structures and to assess that data is undamaged and unaltered in peer-to-peer networks. Dynamo uses Merkle trees for its anti-entropy (replica synchronization) protocol: each node in the system maintains a Merkle tree for each key range it hosts, so as to check whether those keys are synchronized. In this way it is sufficient to just exchange the root of the tree, among the nodes which have to get synchronized, to verify if data is uncorrupted.

Consistency considerations

Finally, in order to provide eventual consistency, which propagates replicas to all nodes asynchronously, Dynamo makes use of data versioning.
Each write operation is treated as a new immutable version of the data, even

if it is just an update, but in this case the system tries an automatic syntactic reconciliation between the old and the new version. In case of a read operation, instead, the client may receive several different versions of the same item; this implies that applications should be designed in order to handle multiple versions of the same datum (semantic reconciliation).
Data versioning is internally implemented with vector clocks, which are $(node, counter)$ pairs associated with each data item. During each write request, the client application sends also a context variable, along with the item to be persisted, which contains the vector clock previously got with a read operation. Upon receiving this version, the node increments the vector clock counter and stores the item.
Since vector clock size may grow too large, Dynamo contemplates a truncation scheme, i.e. when a threshold is reached the oldest version is deleted.

Dynamo influences

Amazon Dynamo has influenced many other NoSQL databases belonging to different categories (e.g. key/value stores, document stores and column-based stores).
For example Scalaris, a key/value NoSQL database written in Erlang, borrows a P2P mechanism, used to expose a distributed hash table to clients, similar to the one used by Dynamo.
CouchDB is a Document-based NoSQL database which takes advantage of a data versioning algorithm that delegates conflict resolution to the client applications in the same way Dynamo does.

As a further example, Cassandra, a column-based NoSQL database, can be considered. Cassandra uses several technologies described by Amazon Dynamo paper such as partitioning, replication, membership and failure-detection algorithms.
Cassandra partitioning algorithm relies on consistent hashing algorithm, but instead of solving load-balancing problems throughout virtual nodes, it employes a measurement and analysis of the load of the various servers in the system to decide where each node should be placed in the ring. Replication, as in Dynamo, is achieved by setting a replication factor r and then distributing data items to the next r nodes in clockwise direction. Finally membership and

failure detection are implemented by means of a gossip-style protocol, similar to the one used by Dynamo, named Scuttlebutt.

2.4 Google App Engine Datastore

This paragraph is about Google App Engine (GAE) Datastore, a schema-less data-storage system. It will cover datastore underlying internals features, its data model and consistency properties. Queries on the Datastore are beyond the purposes of this thesis.

2.4.1 Underlying technology

Datastore is built upon several technologies: Chubby lock-service, Google File System (GFS), Google BigTable and Megastore in order to assure scalability, reliability and performance.
Scalability is achieved thanks to BigTable automatic sharding. Performance are guaranteed by GFS and Chubby which assure a reduced lock granularity and co-location of data. Finally, reliability is function of two technologies: BigTable data replication and Megastore transactions which permit strong consistency over data.

Megastore

All of these features have been described in the previous section, except for Megastore.
According to paper [5], Megastore is a transactional indexed record manager built on top of BigTable which adds richer primitives to it, such as ACID transactions, indexes, and queues.
It introduces the concept of entity groups used to partition the database in order to enable "transactional features for most operations while allowing scalability of storage and throughput". Entities within an entity group are modified by ACID transactions (whose commit record is replicated through Paxos). Megastore uses Paxsos algorithm not only for locking, master election or replication of metadata, but also to replicate user data among different datacenters.

Datastore writes

Before an entity reaches Google server it is encoded into a format called protocol buffer, typically used for remote procedure calls (RPC). Then, the encoded data is sent to the datastore server, which processes the request in two distinct phases: Commit phase and Apply phase.

Commit phase During this phase the server performs two actions

1. It writes data for the entities to the entity group log.
2. It marks the new log entry as committed.

Apply phase During this phase two write operations happen in parallel:

- entity data is written to disk.
- index rows for the entity are written. This operation execution time is directly proportional to the number of indexed-properties contained by the entity.

The write operations returns immediately after the Commit phase, leaving the Apply phase to finish asynchronously. If the commit phase has succeeded but the apply phase failed, the datastore will roll forward to apply the changes to indexes.

2.4.2 Data model

The basic unit of storage in GAE Datastore is the Entity, identified by a key that makes it unique in the system.

Key

The key is composed by different information:

1. Application ID, used to distinguish entities from those of other applications and to make them unique (another user may use the same identifier). Each application has access only to its entities. Datastore API do not allow to retrieve the application id from an entity.

2. The kind is used to logically group entities of the same type in order to categorize them and facilitate queries, furthermore it ensures uniqueness for the rest of the key.

3. The entity identifier is used to distinguish entities of the same kind; it can be a manually assigned string or an automatic generated number.

4. The ancestor path is a hierarchy containing the keys of the entities which are parents of the given one and thus are in the same entity group.

Properties

Entity data is stored inside one or more properties. Each property is characterized by a name and one or more values. The Datastore supports different properties value types from simple ones (string, integers, float, etc.) to collections (List, Arrays, etc.); a complete list can be found on the online documentation page ([1]). Each property can explicitly be declared as indexable or not; the difference resides in the fact that queries can be performed only over indexed properties. Binary and Blob properties cannot be indexed.

Entity Groups

The underlying usage of Megastore allows an user to declare entity groups by means of the ancestor path; queries on entity groups are guaranteed to be strong consistent, i.e. a client application will retrieve the same version of the entities contained in the same partition group.
An entity group is created by declaring an entity to be a descendent of another entity. If the latter does not descend from any other entity, then it is a root entity.
Datastore uses optimistic concurrency to manage transactions. When more than one transactions tries to modify (delete or update) entities inside the same entity group at the same time , the first transaction that commits is the one that will succeed; all others will fail on commit.
Furthermore, Datastore allows transactions to be applied among (at most 5) entity groups; these are called Croos-Group transactions.

2.5 Azure Tables

Azure Tables is a NoSQL datastore which offers structured, massively scalable, storage in the form of tables; it supports an unlimited number of tables, and each table can scale out to store billions of entities representing up to 200 terabytes (if Tables is the only storage service used within the same account).

2.5.1 Underlying technology

Azure storage architecture (which is the same for Blob, Tables and Queues) can be divided in three different layers [6]:

Front-end (FE) layer handles incoming request by authenticating and authorizing them, and then it routes them to a partition server which resides in the Partition Layer. The choice of the partition server to forward requests to is taken by means of a Partition Map cached by each front-end server.

Partition Layer manages the partitioning of Blob, Tables and Queues objects inside streams contained by the stream layer, provides automatic load-balancing in the system, and contains a Paxos lock service similar to Chubby. Each of these objects belongs to one partition, and each partition has one partition server assigned.

This layer regulates what partition is served on what partition server. Automatic load-balancing and partitions assignment is performed by Partition Master servers which check on the overall load of each partition server.

This layer is in charge of providing transaction ordering and strong consistency for objects; furthermore, it caches object data to minimize disk I/O.

Distribution and replicated File System (DFS) Layer or Stream Layer is in charge of storing data on disk, distribute and replicate it across different servers in order to keep it durable. A partition server should access to a DFS server when data needs to be retrieved.

These three layers are all contained by a so called Storage Stamp, a cluster of N racks composed by storage nodes, located on separated fault domains. A

component, named Location Service, manages all storage stamps for disaster recovery and load-balancing. The location service is distributed across two separate geographic locations for disaster recovery purposes.
The stream layer together with the partition layer provide strong consistency at the object transaction layer. Azure Storage uses two different mechanism to replicate data:

Intra-stamp replication (Stream Layer) , a synchronous replication happening at the stream layer and based on Paxos algorithm which creates replicas of the data across different nodes on several fault domains. It provides durability against hardware failure.

Inter-stamp replication (Partition Layer) , an asynchronous replication mechanism which replicates data between stamps. It is used to replicate object in stamps and the transactions on them. Indeed, it provides redundancy against disasters happening in some geographic area.

2.5.2 Datamodel

In order to access to Windows Azure Storage a Storage Account is needed. Each storage account can host up to 200TB of data in total for Blob, Queue and Table services.
Azure Table service is made of Tables which contains Entities; each entity may contain different properties. Since Azure Tables does not enforce a schema on tables, each entity can contain a different set of properties.
An entity is similar to a database row and its maximum size (i.e. the size of all the data in an entity properties) can be 1MB maximum.
A property is a key-value pair, where the key represents the name of the property, for that particular entity, and the value is the actual value stored by the property. Each entity can contain up to 252 properties; futhermore, three more properties are fixed: Row key, Partition Key and Timestamp.
Azure properties support the most common type values: string, integer, boolean, etc. (a complete list can be found on the online documentation [2]); complex data types, such as lists, can be stored as arrays of bytes.

Keys Row keys, inside the same table, must be unique. Row key and partition key, together form a primary key, called clustered index, for the given

entity. The clustered index is first ordered by Partition Key and then by Row key.

Timestamp The Timestamp, instead, represents the date and time the entity was last modified, and it is used internally to provide optimistic concurrency.

Partitions

In order to support load-balancing across storage nodes, tables are partitioned by means of the partition key value.
A partition is formed by a consecutive number of entities, in the same table, possessing the same partition key.
The fact that each partition is served by one partition server permits to have faster responses on queries and strong consistency for entities in the same partition. Anyway, these properties come at the cost of a limit on throughput – i.e. a partition can serve up to 500 entities per second – but, in general, the instantaneous throughput of a partition depends on the load of the partition server.
Entities in the same table, with the same Partition Key, are said to be in the same Entity Group. The entity group permits transactions over contained entities.

2.6 Summary

This chapter discussed NoSQL databases by firstly comparing them to relational ones, in the context of new scalable applications and big data which, thanks to cloud computing, are becoming more and more popular. After having described what choices bring a developer to choose a NoSQL database for his projects, a common NoSQL categorization have been proposed. Two of the most important papers, that have led to the development of almost all NoSQL databases, have been analyzed in order to highlight the common practices used to address several problems when building distributed, scalable and highly available NoSQL databases. The last paragraphs of this chapter concentrated on column-based NoSQL databases that will be used, in this

thesis, to develop a middleware capable of migrating data between columnar databases in an interoperable way.

Chapter 3

Definition of the migration system

3.1 Introduction

The main purpose of this thesis is to design and build a system able to migrate data across NoSQL databases, provided by different vendors. This topic is becoming more and more relevant as the number of NoSQL databases increases. Since NoSQL databases have not been standardized yet, each vendor is building its own NoSQL database solution which is typically incompatible with the others. Furthermore, NoSQL databases are being offered both as standalone system, and as a service (DaaS) by cloud providers.
This plethora of non-standardized NoSQL databases causes a phenomenon called vendor lock-in, i.e. the customer becomes dependent on vendor services and is unable to switch to another vendor without incurring in significant costs. Switching to another vendor may be motivated by several project requirements; for example, some data may need to be modelled differently from other data, or some data may need a different level of consistency, or maybe a more expressive querying language, etc. If all this requirements are not clear, or cannot be established at design time, then the consequences may be catastrophic.
A solution to these problems may be given by 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 would let database user choose the proper service to store his data even after an initial wrong choice, or in the event that a new vendor enters the market offering more adequate solutions for his problems.

In the context of databases, data migration is the process of transferring data between two or more different databases. Typically this task is performed automatically and requires as few inputs from the user as possible.
Data migration is usually achieved thanks to the definition of a metamodel that groups all the common features of the different databases to be migrated. Once this metamodel has correctly been defined, it is possible to describe two translators for each database, that will transform data from the database to the intermediate model and viceversa.
The advantage of this kind of approach is that when a new database has to be supported, just two new mappings rules need to be defined in order to map the data to any other database. In contrast with the approach which imposes to define a mapping rule for any database supported by the system, together with an inverse one.
So, if N is the number of databases supported by the system, while in the first way you would need $2 \cdot N$ mappings (i.e. the number of translators grows linearly), in the second way $N \cdot (N - 1)$ translators would be needed (i.e. the number of translators grows quadratically).
The concept of an intermediate model is also used by Torlone et al. [11] to define a schema translation tool for Web data, whereas, Atzeni et al. propose a common programming interface [4], based on a metamodel previously defined in paper [3], to be used as a library in projects that aim to use different NoSQL solutions.

This chapter introduces to the design and the development of such a system by firstly defining the main requirements the system should respond to. Taking into account all those requirements the system underlying architecture is described. Section 3.4 introduces the metamodel that enables data migration between different NoSQL columnar databases. Finally, a deep analysis of the system design process is conducted. In this phase, some design patterns are adopted in order to build an extensible and reusable system.

3.2 Requirements

We want to design a system capable of migrating data from a columnar database to another.

In the rest of this thesis we will refer to "applications hosted on different cloud services which make use of data that needs to be migrated" as remote applications.
This thesis will concentrate exclusively on complete data switch over, i.e. we suppose that source and destination databases are not utilized during migration phase; in short we can say that the migration happens when remote applications are offline. Future developments, which consider different scenarios, are discussed in Chapter 6.
This system should be independent, whenever possible, from remote applications; hence, it must sit in between and should know nothing about how data is generated. It should be also always available to serve migration needs. Furthermore, it should communicate with databases by means of official libraries, for maintenance and update reasons. Moreover, it should be designed in such a way that developers may easily support new databases; hence, the system should be independent from the implementation of the databases' connectors and should present clear interfaces to the programmers.
Finally, it should take into account different latencies among read and write operations for the diverse databases.

3.3 System Architecture

Given the above requirements, in order to have an independent system always available, a client-server architecture is needed. In particular, the system should reside in an application server (e.g. Tomcat) and should expose at least a method 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. Hence, some methods to handle credentials need to be implemented.
For these reasons it has been decided to implement some REST APIs that allows external applications or users to interact with the system.
A general overview of the system is given in Figure 3.1. The internals of the system have been represented by means of two processes which behave according to the producer-consumer approach described in Section 3.5.2. The queue contains Metamodel objects, whose structure is reported in Section 3.4.

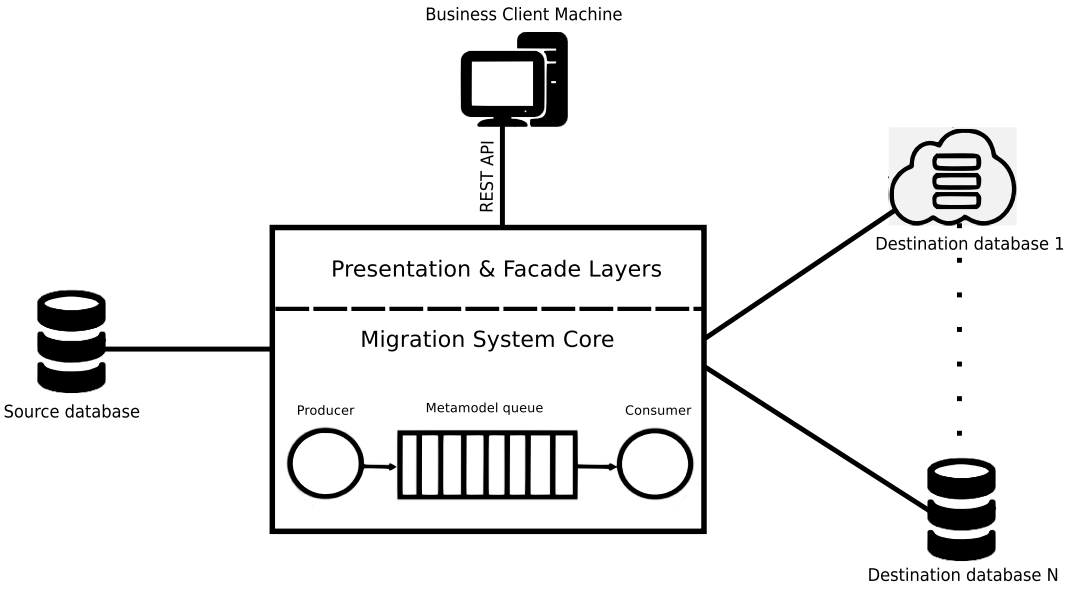

Figure 3.1: Migration System overview

The client-server system architecture is multi-tiered – i.e. system functionalities are logically separated across different layers – in order to provide a more scalable and reusable system. System architecture is shown in Figure 3.2.

Representation Layer This layer is the system entry point for users and external applications (clients), i.e. it exposes REST API to let client applications interact with the system. Its response formats are JSON or XML depending on the Content-Type of the request message. REST API design is discussed in Appendix A.
This layer communicates directly with Business Client Facade and with Credentials Access Layer by means of standard method calls.

Business Client Facade This layer contains the interfaces needed to request database switch over to the lower layers. Moreover, it holds a generic abstraction for the supported databases, which describes their common operations, e.g. connection, disconnection, etc.
More details about the interfaces design process will be given in Section 3.5.

Business Layer This layer groups the business logic used to perform the actual migration between databases. The logic consists of two threads, a producer and a consumer which, respectively, extract data from the source

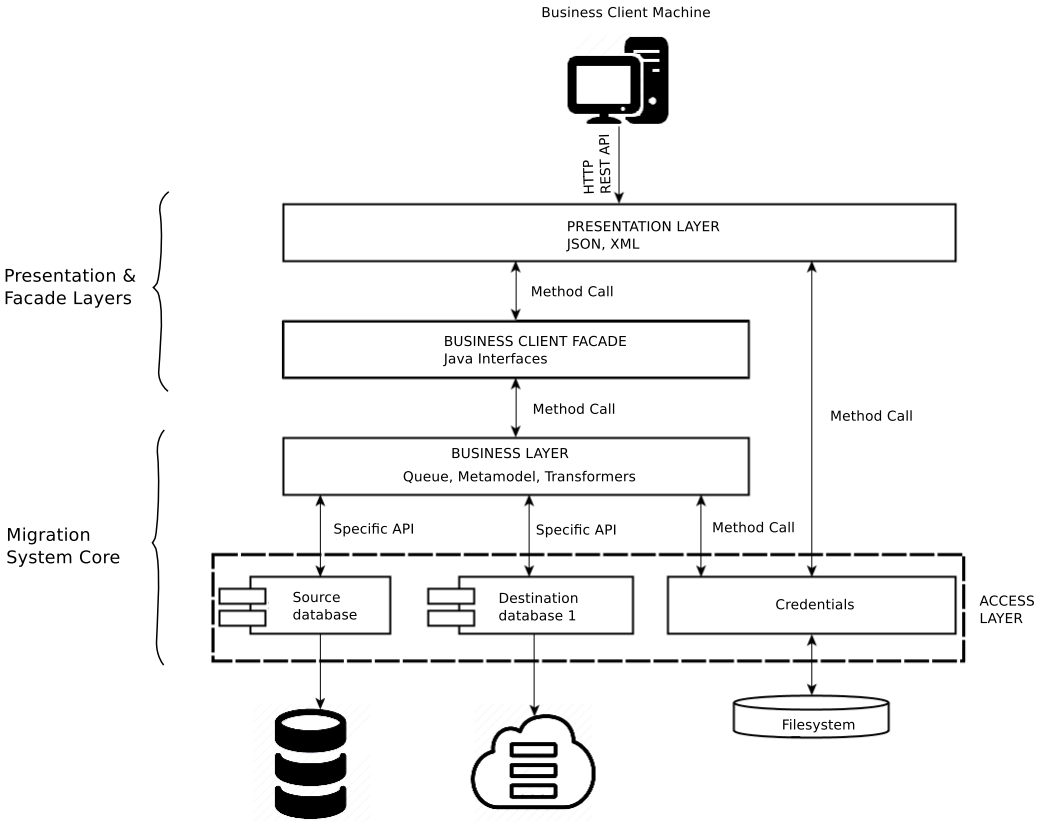

Figure 3.2: System architecture

database and put them in the destination database. The two threads communicate by means of a common shared queued, more details are provided in Section 3.5.2. Hence, this layer contains also the intermediate Metamodel and two transformers for each database. Furthermore, this layer directly communicates with the Access Layer which allows to actually retrieve and store information, by means of library specific API.
Metamodel design is discussed in Section 3.4, whereas Transformers design is treated in Chapter 4.

Access Layer This layer is in charge of performing read and write operations by means of different protocols.

Databases Typically database libraries uses different API that, after manipulating data passed to them, contact the respective database services in order to transfer data through HTTP protocol.

Credentials Credentials are stored directly, in the form of key-value pairs, inside a file in the local file system.

3.4 Metamodel design

As stated in Chapter 2 the study is restricted on column-oriented NoSQL databases. By analyzing the various type of columnar databases it is evident that their data model derives from the one proposed in BigTable paper. So, in order to perform a complete mapping between different 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 with the switch over are strong consistency and secondary indexes (for databases that support them). In the case in which the source database supports secondary indexes, whilst the destination database does not, auxiliary data structures, preserving indexed properties, are created in the destination database by the migration system. The choice on how to design these data structures depends on the destination database, hence, it is delegated to the specific database translator, as described in Chapter 4.
To the best of our knowledge, the metamodel presented in this thesis 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 3.3 has been designed; it is located in the Business Layer, and it can only be accessed by methods residing in the same layer. A detailed description of each Metamodel component is given in the following paragraphs.

Column The basic unit for storing data properties is the Column. It contains the single datum property value to be persisted along with its name; furthermore, it provides an explicit way to declare the datum property type and if the property should be indexable or not.
Column name field, of type String, should contain printable datum property name.
The system provides standard utility methods to serialize (deserialize) data

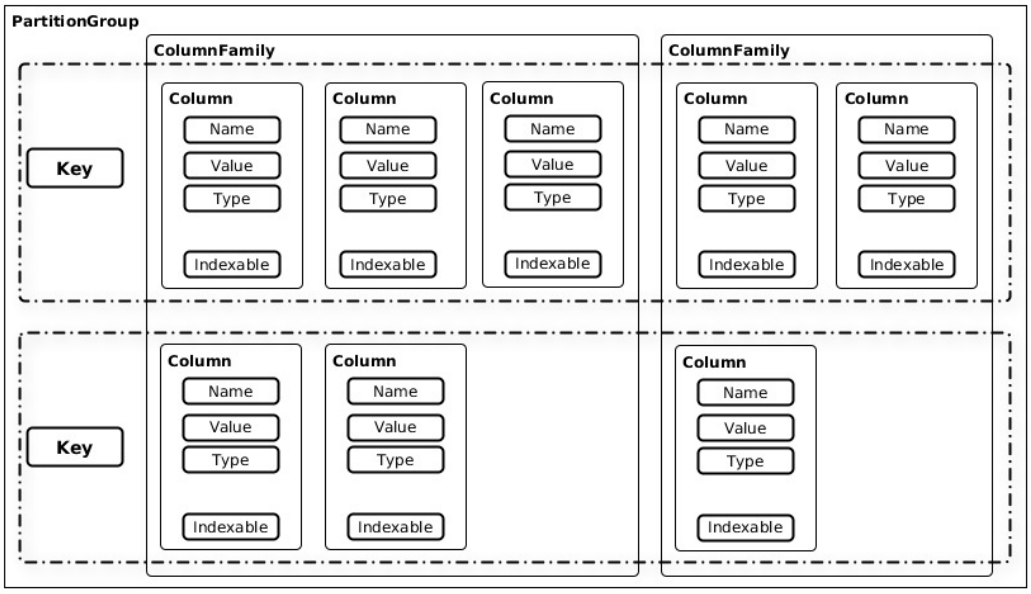

Figure 3.3: Metamodel

into (from) Column value field, which should contain data in the form of bytes array.
Column type field should contain datum property printable type that will be used to deserialize data inside Column value field, if the destination database supports its original data type. Otherwise data will be stored as an array of bytes (typically supported by any database).
Column indexable field contains a boolean value which states if data contained in Column value field should be set as indexable or not in the destination database (if it supports indexes).

Key A set of Columns referring to the same datum is named Entity. Entity Keys are used to index Entities.

Column Family As stated in Google BigTable paper, Column Families are just groupings of different Entities Columns of the same type . 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 data scheme is not fixed – Column Families may

contain diverse number of Columns for every Entity, whose Columns belong to the same Column Family.

Partition Group Some databases provide strong consistency by simply letting users model data in constructs specific for each database, e.g Google Datastore uses ancestor paths, whereas Azure Tables uses a combination of Partition Key and Table Name.
For this very reason the Metamodel provides the Partition Group; i.e. Entities inside the same Partition Group are guaranteed to be stored in the destination database, in such a way that strong consistency will be applicable to them on every operation performed by the destination database.

3.5 System design

Requirements expressed in Section 3.2, request that the system should support several databases and use their official connectors. For this reason, a common programming language, able to support the majority of the databases connectors, should be used. Hence, Java has been chosen.
Furthermore, the system should expose clear interfaces to the programmers and abstract the general parts of it from the database drivers logic. In order to do so, three different design patterns have been adopted. The first two of them reside in the Business Client Facade, since they are used to abstract databases logic and perform the switch over. Whereas, the transformer pattern has been used for classes hosted by Business Layer, since they are part of the business logic which performs the actual mappings.
In software engineering design patterns are used to solve problems that recur, with some slightly variation, when building an application. They propose standard solutions that help to improve software quality and reduce development time, making the code extensible and reusable. Design patterns are in contrast with custom solutions in that, the latter may lead to more complex or inefficient implementations for a given problem and they may result in code which is harder to debug, extend and reuse.
For these reasons, a deep insight of how design patterns are used for building this system is reported in Paragraph 3.5.1.
Finally, latencies among different operations and databases are mitigated by

means of the producer-consumer approach, i.e. two concurrent processes share a queue containing Metamodel objects. Further details are reported in Paragraph 3.5.2.

3.5.1 Design strategies

In order to build a reusable and extensible migration system, several design patterns have been adopted. Design patterns were originally classified based on the type of problem they aimed to solve, i.e. creational patterns, behavioural patterns and structural patterns; recently new patterns and classifications have emerged, but they are not needed in this system.
Three patterns, described in the following paragraphs, have been tightened together, to accomplish previously described tasks, and form a so called pattern language. After a brief introduction, each paragraph describes how each pattern is used in order to build the system depicted in Figure 3.4.

Defining common migration procedure

The migration system needs to implement a common mechanism to perform data migration from a source database to a destination one. In order to do so, several atomic steps – e.g. connection, translation, etc. . . – should be executed in a given order, independently from the underlying database technology. For this reason we use the template pattern, which belongs to the class of behavioural patterns, and focuses on the interactions between objects.
In particular, the template pattern defines the steps of an algorithm and the order in which these steps should be executed, but lets subclasses implement some of them.
These steps are implemented using abstract methods inside an abstract class. Classes that will inherit the abstract class will not be able to override the abstract methods, if they use the final modifier. The abstract class may also define hook methods that can be overridden by subclasses.
This is exactly what happens for class AbstractDatabase in Figure 3.4. In that class, hook methods are declared – i.e. connect, disconnect, fromMetamodel, toMedamodel – together with the abstract method switchOver. The latter method implements the switch over procedure between two databases by using a combination of the hook methods.

Figure 3.4: UML class diagram

When a new database needs to be supported by the system, it will be sufficient to extend AbstractDatabase and override the hook methods.
Hook methods contain the logic specific to each database and depend on the database connector.

Handling different databases interoperably

In order to build a migration system which is as general as possible, some precautions should be taken. In particular, the system should not be aware of the underlying database implementation, and should be able to operate with any kind of source and destination database without any further knowledge. Hence, an abstract way to deal with different databases is needed.
Creational patterns offer an interface which exposes methods needed for instantiating objects, in order to mask how objects are actually implemented.
Factory pattern is a member of creational patterns and it provides an interface to create an object without exposing the creation logic to the client. Furthermore, it lets the subclasses decide which object to instantiate and then refer to it using a common interface.
In this specific case, Factory Pattern is used to seamlessly instantiate source and destination databases, and call common methods (such as switchOver) without warring about the actual database on which the call is performed. In particular, the class DatabaseFactory is in charge of creating the proper database instance, given its name. Each supported database should implement a common interface, but, since this pattern has to be used together with template pattern, it has been chosen to use an abstract class instead of an interface. Hence, databases classes need to extend AbstractDatabase and implement its hook methods in order to be supported.

Transforming databases data models

The migration system should be designed in such a way to allow developers to easily add support for a new database, without the necessity to modify system code. Hence, the system should provide a "contract" that needs to be agreed on by any newly supported database. For this reason, the system should expose clear interfaces to the programmers.
Thus, we recur to the translator pattern, which is not an actual existing pattern, but it is derived from Message Translator pattern described in book [10].

It consists of an interface which declares methods to transform an object into another format and viceversa.
In particular, the interface ITransformer exposes a method to convert a Database model object to a Metamodel object, and a method to convert a Metamodel object into a Database model object. For each new database that needs to be supported, a transformer class, implementing these two methods, needs to be defined. The implementation of the database model depends on the specific database, hence it is not fixed. Whereas, the Metamodel design has been analyzed in Section 3.4.
Thus, each class extending AbstractDatabase uses a Transformer object inside the hook methods toMetamodel and fromMetamodel to provide the actual transformations. Tranformers design and implementation will be covered in Chapter 4.

3.5.2 Producer-Consumer approach

Distributed database architectures have different read and write latencies; typically write operations are slower than reads. Hence, in order to migrate data, it is unfeasible to extract data (convert them) and directly write converted data into the destination database. Because, by doing so, read request from the source database may time out and elements successive to the current one may not be retrieved.

A solution to the above problem is given by the producer-consumer approach: two processes share a common buffer which is used as a queue, as shown in Figure 3.1. The producer should not consume data when the buffer is full and the consumer should not read from the queue when it is empty.
Java provides an implementation for this queue, called BlockingQueue, which is thread-safe and offers automatic concurrency control.
All the classes and interfaces described reside in the Business Layer.
The template method switchOver, contained in AbstractDatabase class, creates two threads: a producer and a consumer, according to the approach described above. Both of them use the hook methods to perform connection and disconnection operations. A sequence diagram illustrating their behaviour is shown by Figure 3.5.

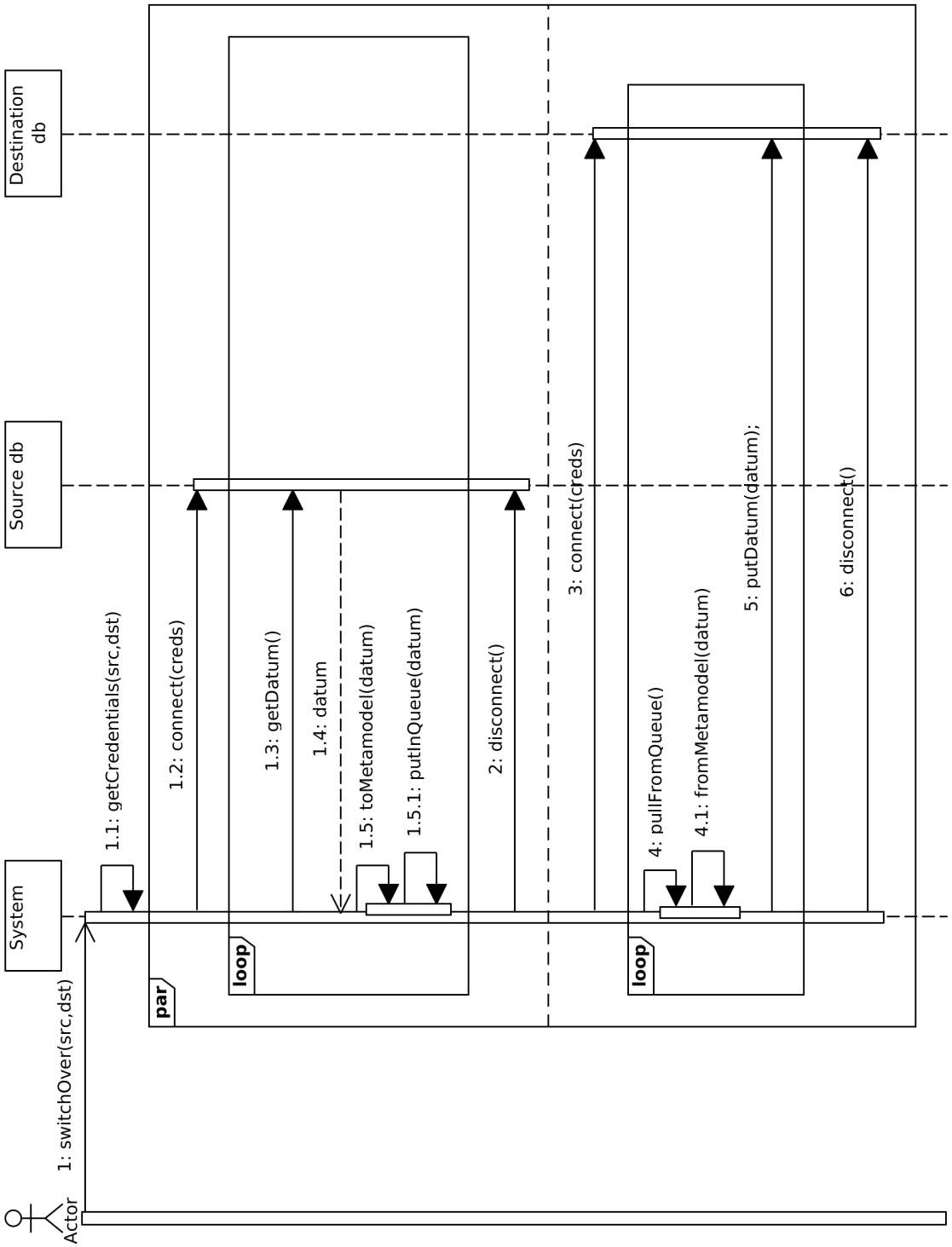

Figure 3.5: Switch over sequence diagram

The producer extracts data from the source database which are passed to the hook method toMetamodel and stores the resulting Metamodel object inside the blocking queue.
The consumer waits for an object to be inserted inside the blocking queue. Once the consumer notices that at least an element is present inside the queue, it pulls the element which is then passed to the hook method fromMetamodel, and sends the resulting object to the destination database that will store it.
The process stops when the producer has extracted all the elements from the source database and the consumer has pulled and converted all the Metamodel objects inside the queue.

3.5.3 Connection and Disconnection management

Figure 3.5 shows how the system transparently communicates with source and destination databases despite the fact that libraries and underlying technologies are completely different. In particular, connection and disconnection methods rely on those proprietary libraries to operate, but, in order to build a system that is as general as possible, these methods need to be wrapped by components implemented following a standard approach. These components represent the Access Layer of the system, as shown in Figure 3.2.
During the connection or disconnection phase, in case of an error all exceptions are caught and a new custom exception (ConnectionException) is created and then thrown; this allows to have a standardized connection mechanism for all the supported databases. In this way, it suffices to insert the hook connect method inside a try-catch block and just the ConnectionException will be thrown, so that it can be handled in the same way by all the databases.
The disconnect method can be called inside the finally block, so that other operations do not have to care about connection and disconnection management.
After having established the connection, an object used for operating with the database is typically returned; this object is used across the system to keep track of the status – i.e. if it is instantiated the system is connected to the given database; if it is null then the system has not established a connection yet. A check method is implemented and it can be called by other methods to verify the status of the connection.

3.5.4 Credentials management

Usually, one needs to provide some kind of credentials to databases in order to retrieve and store data. For example in Google App Engine (GAE) Datastore calls can be made providing the same username and password needed to access all GAE services; whereas, Azure Tables requires a particular hash string generated from its online configuration portal.
In order to provide a standard interface for programmers to handle credentials, a class named PropertiesManager has been implemented. This class, together with those handling databases, is part of the Access Layer. It exposes utility methods to create, read, update and delete credentials, in the form of key/value pairs stored in a file. Furthermore, it implements other utility methods and a nested class to make those operations above accessible even by REST API (more details can be found in Appendix A).

3.6 Summary

This chapter proposed a solution for the design and implementation of a system for data migration between different NoSQL columnar databases.
After an overview of the system architecture, the chapter presented an original Metamodel to enable translations between different NoSQL databases and to preserve, whenever possible, properties like strong consistency and secondary indexes. Finally, a deep analysis of the system design phase was conducted, highlighting the main design choices taken, in order to build reusable code that allows to seamlessly plug in new components, so as to support new databases.

Chapter 4

Metamodel Transformations

4.1 Introduction

As reported in Chapter 3, translators have been designed according to the translator pattern. It is sufficient to instantiate a transformer class from a hook method (contained by classes that extend AbstractDatabase) to operate the translations.
In the following sections, we will demonstrate how to add support for a new database by showing the general guidelines to design translators.
Section 4.2 studies direct translators, i.e. mappings between a source database and the Metamodel defined in Chapter 3. Whereas, Section 4.3 covers inverse translators, i.e. mappings from the Metamodel to a destination database.
For each translator type, two practical cases will be covered: GAE Datastore and Azure Tables; and, for the solely purpose of guiding throughout the reading of the examples sections, some pseudo-code excerpts are reported.

4.2 Direct Translators

A direct translator converts data from a source database into the Metamodel representation.
The basic approach to create a correct mapping is to identify the single datum container – e.g. a tuple in relational databases or an entity both in Datastore and Azure Tables. This element can be mapped to a Metamodel entity and data contained in it can be translated into Columns.
At this point it is fundamental to extract the proper information. Name field is straightforward, whereas the value field should contain serialized data, since it

is not possible to know in advance if a type is supported by all other databases. Hence, the system provides a utility method to perform serialization; it will be sufficient to pass the value to this method to generate a valid, serialized, byte array to be stored inside value field.
The type field should contain a string which states the type of the object previously serialized. It typically can be obtained throughout reflection on the object previously serialized.
If the source database supports secondary indexes then it is easy to establish if a column should be set as indexable or not. Otherwise, the translator should have knowledge about what type of queries will be expressed over data. A trivial solution may be that of setting all columns as indexable.
The choice of a proper mapping for the Column Family depends on the source database, but in general its purpose is to group Columns of the same type. Under certain circumstances, it may happen that the Column Family will contain an entire entity.
Finally, the Partition Group is a construct used to preserve strong consistency and, if the source database supports it, the implementation may vary for each database. In general, the Partition Group should contain all the entities for which transactionality is desired. Since the Metamodel represents the Partition Group as a string, it suffices to set the same value to two different entities, in order for them to be accessed in a strongly consistent way.
In all NoSQL databases encountered so far, transactionality may be seen as a function of two elements, hence a non-restrictive convention on the Partition Group string has been adopted, i.e. the string is divided into two parts by an hash symbol, e.g. `<name>#<value>`.
Practical examples of direct database translators are shown in the following subsections.

4.2.1 GAE Datastore

The Datastore class, implementing AbstractDatabase class, extracts all information entity by entity. In order to do so, as can be seen also in Algorithm 1:

1. all Kinds present in the Datastore should be retrieved; this is achieved by means of metadata queries.

2. Subsequently, all entities for a given Kind are extracted.

Algorithm 1 Datastore Class excerpt

Precondition: Implements AbstractDatabase class

```
1: kinds ← datastore.getKinds()
2: for each kind in kinds do
3:     entities ← ds.getEntities(kind)
4:     for each entity in entities do
5:         metamodelEntity ← DatastoreTransformer.toMetamodel(entity)
6:         Queue.put(metamodelEntity)
```

Table 4.1: Datastore – Direct Mapping

Datastore	**Metamodel**
Entity	Entity
Entity Key Identifier	Entity Key
Entity Kind	Column Family
Property Name	Column Name
Property Value	Column Value (serialized)
Property Value type	Column Value type
Property.isUnindexedProperty	Column Indexable
Ancestor Path root Entity	Partition Group

3. Then, each entity is passed to the transformer class, which is in charge of analyzing the entity and properly map it to the Metamodel.

A summary of the mappings is shown in Table 4.1.

Algorithm 2 Datastore Transformer Class excerpt

Precondition: metamodel object has already been instantiated

```
1: function toMetamodel(entity)
2:     rootEntity ← generateRoot(entity)
3:     mapPartitionGroup(metamodel, rootEntity)
4:     mapKey(metamodel, entity.key)
5:     mapColumnFamily(metamodel, entity.kind)
6:     mapColumns(metamodel, entity)
7:     return metamodel
```

Mapping functions, reported in Algorithm 2, are investigated in the following paragraphs.

Entity Key and Column Families mapping

1. Datastore entity key identifier is directly mapped to Metamodel entity key.

2. Datastore entity Kind can be mapped to a Metamodel entity column family, since it is used to group Datastore entity properties.

Notice that, with this mapping, all Datastore entity properties will reside in just one Metamodel Column Family.

Entity Column mapping Datastore entity properties are equivalent to Metamodel entity columns.
In fact, the following mappings apply:

1. Property name can be directly mapped to Column name.

2. Property value type is mapped to Column type and it is derived from Datastore Property object, contained by the Entity object, by means of reflection.

3. Property value is directly serialized into a byte array and then stored in Column value.

4. Finally, since the Datastore provides secondary indexes, a simple call to Property method `isUnindexedProperty` returns a boolean value which is stored by Column indexable field.

Partition Group mapping The ancestor path is the mechanism which guarantees transactionality in the Datastore and it is uniquely identified by the root entity, i.e. all entities in the same ancestor path will descent from the same root entity. Hence, by determining root entity Kind and Key it is possible to properly assign a Partition Group to any entity.

Root entity extraction A root entity can be extracted from any Datastore entity key (even though it is not documented) by requesting a string representation of it. This request will return the whole ancestor path which is of the form `Kind(Key)/Kind(Key)/...`; by extracting the first occurrence of the pair `Kind(Key)`, root entity Kind and Key can be derived.

Root entity mapping The Partition Group named after the extracted root entity is generated. Its name is `Kind#Key`, and it is unique for that specific ancestor path.
If entities, inside the Datastore, do not declare a parent entity, then each entity will be mapped to a different Partition Group.

Considerations Google App Engine Datastore allows to express queries over descendent entities, i.e. ancestor paths are treated like relations among entities. This characteristic gets lost during the direct mapping procedure.
All things considered, since Datastore is the only database of its kind (i.e. Column-based) to provide relations among data, Metamodel direct translation for this database can be considered acceptable.

Even though the Datastore offers other documented mechanisms to reconstruct an ancestor path, they are not as performant as the method described in the previously.
The first approach consists in requesting all descendents for a given entity. This way of proceeding requires to make a new query for every entity that needs to be mapped; it can be extremely inefficient for two different reasons:

1. Each query involves considerable latencies. Hence, the overall switch over performance would be influenced.

2. Since an entity can be a root one or not, in order to reconstruct a whole ancestor path, it may be necessary to store a huge amount of entities in memory, before determining the correct relations among all of the entities.

The second approach may exploit Datastore statistics queries in order to firstly retrieve all root entities and then derive all descendents. This approach suffers of a problem as well: statistics queries operate on data which get computed automatically by the Datastore once or twice in a day, hence results retrieved by this queries may not be always reliable.

A third approach would force remote applications to add a property to each entity, which states if that it is root or not. Hence, Statistics queries wouldn't

Table 4.2: Azure Tables – Direct Mapping

Azure Tables	**Metamodel**
Entity	Entity
Row Key	Entity Key
Table	Column Family
Property Name	Column Name
Property Value	Column Value (serialized)
Property Value type	Column Value type
-	Column Indexable
Table name + Partition Key	Partition Group

be needed, since it would be possible to retrieve all root entities with just a query, and the previous approach problem would be solved. Unfortunately, the system wouldn't be transparent from remote applications point of view and it would depend on remote application implementation.

4.2.2 Azure Tables

Algorithm 3 Tables Class excerpt

Precondition: Implements AbstractDatabase class

```
1: tableList ← azure.getTableList()
2: for each table in tableList do
3:     entities ← azure.getEntities(table)
4:     for each entity in entities do
5:         metamodelEntity ← TablesTransformer.toMetamodel(entity)
6:         Queue.put(metamodelEntity)
```

The Tables class, implementing AbstractDatabase class, extracts all information entity by entity. In order to do so, as can be also seen in Algorithm 3:

1. All tables' names present in Azure Tables are retrieved.
2. All entities contained by the same table are extracted.
3. Each entity is passed to the transformer class, which is in charge of analyzing the entity and properly map it to the Metamodel.

A summary of the mappings is shown in Table 4.2.

Mapping functions, reported in Algorithm 4, are investigated in the following paragraphs.

Algorithm 4 Azure Tables Transformer Class excerpt

Precondition: metamodel object has already been instantiated

```
1: function toMetamodel(entity)
2:     mapKey(metamodel, entity.rowKey)
3:     mapColumnFamily(metamodel, entity.table)
4:     mapColumns(metamodel, entity)
5:     mapPartitionGroup(metamodel, entity.table, entity.partitionKey)
6:     return metamodel
```

Entity Key and Column Families mapping

1. Azure Tables entity Row Key is directly mapped to Metamodel entity key.

2. Azure Tables entity table can be mapped to a Metamodel entity column family, since it is used to group Azure Tables entities.

Entity Column mapping Azure Tables entity properties are equivalent to Metamodel entity columns.
For this reason the following mappings can be applied:

1. Property name can be directly mapped to Column name.

2. Property value type is mapped to Column type and it is derived from Azure Tables `EdmType` object, contained by each EntityProperty object.

3. Property value is directly serialized into a byte array and then stored in Column value.

4. Finally, since Azure Tables does not provide secondary indexes, if the source data type is not an array then Column indexable field is set to true.

Partition Group mapping Strong consistency in Azure Tables is achieved by a combination of the Row Key and Table name, i.e. for all entities having the same Partition Key, and contained by the same table, transactions may be defined. Hence, by determining entities Partition Key and Table name, it is possible to properly assign a Partition Group to any entity.
Since Partition Key and Table name can simply be extracted from an entity, the Partition Group named `Table Name#Partition Key` can easily be

generated.

4.3 Inverse Translators

An inverse translator converts data from the Metamodel into the destination database representation.
The basic approach to create a correct mapping is to identify the destination database single datum container in the same way of Section 4.2. This element can be populated from a Metamodel entity. The inverse translation strongly depends on the destination database data model. Other key destination database aspects to be considered are: supported data types, consistency properties and secondary indexes.
As regards data types, Metamodel Column value always contains serialized data; if the destination database supports the data type contained in Metamodel Column type field, then the value can be deserialized by means of the utility method provided with the system. Otherwise, the value is kept serialized and a new "property", in the destination database, should be created in order to store the original data type. Direct translators should check for such a "property" whose name is, conventionally, composed by "`<Column_Name>#Type`".
If the destination database supports indexes, then during the inverse translation stage, data item should be marked as indexable; obviously the procedure changes with the database.
If the destination database does not support secondary indexes two approaches can be taken: do not consider information regarding secondary indexes, or store that information in auxiliary data items inside the destination database. The latter approach should be chosen in case data, that has already been migrated, should be transferred back to the original database or migrated again to a another database which supports secondary indexes.
Finally, entities in the same Partition Group should be stored, by the destination database, in such a way to guarantee strong consistency. Not all databases permit strong consistency and when they do, it is platform dependent; hence, examples of two databases, that support transactionality, will be provided in the following paragraphs.

Table 4.3: Datastore – Inverse Mapping

Metamodel	**Datastore**
Entity	Entity
Entity Key	Entity Key Identifier
Column Family	Entity Kind
Column Name	Property Name
Column Value	Property Value (deserialized)
Column Value type	Check on supported Property data types
Column Indexable	Property set index
Partition Group	Fictitious root entity

4.3.1 GAE Datastore

Algorithm 5 Datastore Class excerpt

Precondition: Implements AbstractDatabase class

```
1: while true do
2:     metamodelEntity ← Queue.take()
3:     datastoreEntities ←
           DatastoreTransformer.fromMetamodel(metamodelEntity)
4:     datastore.put(datastoreEntities)
```

The Datastore class, implementing AbstractDatabase class, performs inverse translation by applying the following steps, as can also be seen in Algorithm 5:

1. Extracts Metamodel Entities from the queue.
2. Each entity is then passed to the Datastore translator that performs the correct transformation from Metamodel representation to Datastore data model.
3. When a proper Datastore Entity is returned from the transformer, it gets persisted to the Datastore.

A summary of the inverse mappings is shown in Table 4.3.

Mapping functions, reported in Algorithm 6, are investigated in the following paragraphs.

Algorithm 6 Datastore Transformer Class excerpt

Precondition: datastoreEntities, a List of Datastore Entities, has already been instantiated

```
1: function fromMetamodel(metamodelEntity)
2:     if !existsFictitiousEntity(metamodelEntity.partitionGroup) then
3:         fictitiousEntity ←
               createFictitiousEntity(metamodelEntity.partitionGroup))
4:     for each columnFamily in metamodelEntity.columnFamilies do
5:         datastoreEntity ←
               new Entity(columnFamily, metamodelEntity.key,
                       fictitiousEntity)
6:         mapColumns(datastoreEntity, metamodelEntity)
7:         datastoreEntities.add(datastoreEntity)
8:     return datastoreEntities
```

Partition Group mapping Before creating a Datastore Entity, aspects regarding transactionality should be considered. Entities in the same Partition Group should be stored in such a way that the Datastore can apply strong consistency policies on them.
The mechanism which grants transactionality in the Datastore is the ancestor path which is uniquely identified by the root entity.

Fictitious Entity creation Hence, in order to keep transactionality aspects during the migration process, a fictitious root entity, for each partition group, should be created.
Since Metamodel Entities are processed singularly, for performance reasons and to keep as fewer elements as possible in memory, it is not possible to know in advance all entities belonging to the same Partition Group; hence, it is necessary to make a query to the Datastore asking if a fictitious entity, derived from the Partition Group, is already present. If it is not, than it should be added.
Fictitious Entity is generated from Partition Group name. The substring on the left hand-side of symbol "`#`" is concatenated with the symbol "`@`" and mapped to the fictitious entity kind. Whereas, the substring on the right hand-side of symbol "`#`" is mapped to the fictitious entity key identifier.

Datastore Entity creation At this point, a Datastore Entity can be created and its parent Entity will coincide with the fictitious root entity.

Entity Key and Column Families mapping

1. For every Column Family contained by a Metamodel Entity a new Datastore Entity is generated, hence each Column Family is mapped to a Datastore Entity Kind.

2. Every Metamodel Entity Key is directly mapped to a Datastore Entity Key identifier.

This procedure may generate different Datastore Entities, starting from just a Metamodel Entity, with the same key identifier, but different Entity Kinds.

Entity Column A Metamodel Entity may contain several Columns, each of them is mapped to a Datastore Entity property. Aspects that need to be considered in this phase, are secondary indexes and supported data types.
These are the steps that need to be done in order to correctly map a Column to a Property:

1. Column name is mapped to Property name.

2. Before deserializing data contained by Column value field, a check on Datastore supported data types is performed.

    - If that particular data type, contained by Column type field, is supported, then Column value field is deserialized and mapped to the property value.
    - Otherwise, it is kept serialized and stored inside Column value field. Hence, an auxiliary property named `<Column_Name>#Type` is created and the original data type, contained in Column type field, is inserted into it.

3. If Column indexable field contains a true value, then the respective property on the Datastore should be set as indexable too.

Considerations All things considered, in order to preserve strong consistency, in the worst case, one read and two write operations per Metamodel Entity are performed on the Datastore. This can result in significant migration time interval, depending on the direct Metamodel translation.

Table 4.4: Azure Tables – Inverse Mapping

Metamodel	Azure Tables
Entity	Entity
Entity Key	Entity Row Key
Column Family	All stored inside the same table
Column Name	Property Name
Column Value	Property Value (deserialized)
Column Value type	Check on supported Property data types
Partition Group	Table Name and Entity Partition Key

4.3.2 Azure Tables

Algorithm 7 Azure Tables Class excerpt

Precondition: Implements AbstractDatabase class

```
1: while true do
2:     metamodelEntity ← Queue.take()
3:     tablesEntity ←
           TablesTransformer.fromMetamodel(metamodelEntity)
4:     tables.createTable(tablesEntity.tableName)
5:     tables.put(tablesEntity)
```

The Tables class, implementing AbstractDatabase class, performs inverse translation by applying the following steps, as can also be seen in Algorithm 7:

1. It extracts Metamodel Entities from the queue.
2. Each entity is then passed to the Tables translator that performs the correct transformation from Metamodel representation to Azure Tables data model.
3. When a proper Tables Entity is returned from the transformer, it is persisted to Azure Tables.

A summary of the inverse mappings is shown in Table 4.4.

Mapping functions, reported in Algorithm 8, are investigated in the following paragraphs.

Algorithm 8 Azure Tables Transformer Class excerpt

```
1: function fromMetamodel(metamodelEntity)
2:     tablesEntity ← setTableName(metamodelEntity.partitionGroup)
3:     for each columnFamily in metamodelEntity.columnFamilies do
4:         mapColumns(tablesEntity, columnFamily,
               metamodelEntity)
5:     return tablesEntity
```

Partition Group mapping An important aspect to take into account when migrating to Azure Tables is strong consistency. This database offers transactionality to entities contained in the same table and that have the same Partition Key.
Since in the Metamodel representation the unit of transactionality is represented by the Partition Group, a proper mapping between these aspects should be provided.
As stated in previous Section, the convention typically adopted is to separate Partition Group name into two parts, separated by the "`#`" symbol.

- The substring on the left-hand side of the "`#`" symbol is mapped to the Table name;
- the substring on the right-hand side of the "`#`" symbol is mapped to the Partition Key.

In this way, for all entities contained by the same Partition Group, strong consistency can be guaranteed.

Entity Key and Column Families mapping

1. Column Families indexed by the same Entity Key, and contained by the same Partition Group, are all put in the same Table, given that transactionality should be also applied to them.
2. Metamodel Entity Key is mapped to Tables Entity Row Key.

Notice that, the concept of Column Family has no direct counterpart in Azure Tables.

Entity Column A Metamodel Column can be directly mapped to a Tables Property.

1. Column name is copied inside Property name field.
2. Before deserializing data contained in Column value field and storing it as a Property value, a method that performs a check on Azure Tables supported data types, is called. So:
   - if the data type, contained inside Column Type field, is supported by Azure Tables, then Column value is deserialized into the Property value.
   - Otherwise, data is left serialized and copied inside Property value field. And an auxiliary property named `<Column_Name>#Type` is created to store the original data type, contained in Column type field.

Considerations Azure Tables does not support secondary indexes. At the actual stage of the implementation of the system, it has been decided not to create a new table to store secondary indexes information. This choice has been motivated by the fact that, currently, the system supports just two databases.

4.4 Summary

This chapter discussed how different database data models can be mapped to the Metamodel defined in Chapter 3, and viceversa. Approaches adopted take into account databases specific properties and differences, trying to support all of them during the migration process. Moreover, several considerations, regarding approach tradeoffs, have been expressed.

Chapter 5

Migration System Evaluation

5.1 Introduction

In this Chapter several evaluations are reported.
In Section 5.2, we conduct some tests aimed at verifying the performance of the the migration system. In order to do so, we refer to an application, called MiC.
Subsequently, in Section 5.3, we make some further considerations on the migration system, in particular we compare the Metamodel approach with a direct database-to-database translation.
In Section 5.4, we show how a Key-Value database, like Amazon DynamoDB, can be supported by the migration system described in the previous chapters. Moreover, since DynamoDB deals with data consistency in a particular way, several interesting mappings, that exploit the Metamodel properties, are reported.
Finally, in Section 5.5, we draw the conclusions about performed tests.

5.2 Performance and overhead of the migration system

In order to test the migration system described in the previous chapters, data stored by an application called "Meeting in the Cloud" (MiC) has been used. This application, in turn, has been originally developed to test a library, called "Cloud Portable Independent Model" (CPIM) [9].
CPIM aims to build an abstraction layer that enables applications, deployed in different Cloud platforms, to use a common set of API to interact with

any Cloud service supported, independently from the Cloud platform vendor. In particular, the services supported by CPIM are: Blob service, NoSQL service, SQL service, TaskQueue service, MessageQueue service, Memcache and Mailing services. At the moment of writing this thesis, CPIM supports Java applications developed on Google App Engine and Windows Azure platforms.
CPIM NoSQL service exploits Java Persistence API (API), specific for each database implementation, to operate on NoSQL databases.
MiC is a social networking web application, written in Java, that uses CPIM library to interact with all the services listed before. Upon registration, MiC lets users declare their interests by choosing among seven default categories (called Topics). For each chosen category, the user has to answer to some questions, by expressing a rate from 1 to 5 (UserRatings). For these two kinds of interactions –i.e. topics selection and user ratings – information are stored inside the NoSQL database (GAE Datastore or Azure Tables).
MiC then calculates affinities between users and get them in touch, by evaluating the Pearson's coefficient. From this moment on, users will be able to post messages about selected topics. In order to perform these last tasks, MiC uses other services among those listed before, but since they do not affect NoSQL databases behaviour, they are not further investigated by this thesis.

MiC uses a fixed data model that can be represented by the class diagram in Figure 5.1, both for GAE Datastore and for Azure Tables.

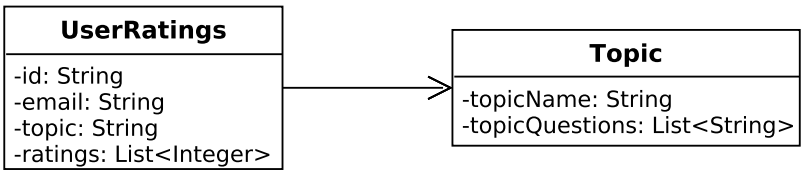

Figure 5.1: MiC NoSQL service Class diagram

- Topic Class stores the topics about the questions that the user answers to. Its attributes are:
  - topicName: which contains the name for the given topic and it is also the unique key.

  - topicQuestions: which contains a list of questions for a specific topic.

- UserRatings class represents the answers that a specific user has given to the questions proposed to him. Its attributes are:
  - id: which uniquely identifies an answer. It is the concatenation of the "email" and the "topicName" attribute – e.g. `email_topicName`.
  - email: which stores user email address.
  - topicName: which represents the name of the topic related to the answers given by the user.
  - ratings: which contains the answers to the questions asked to user "email", regarding topic "topicName".

The following paragraphs show how the two classes in Figure 5.1 have been mapped to the respective databases by CPIM library.

Google App Engine Datastore Each class, represented in Figure 5.1, corresponds with a Kind, respectively UserRatings and Topic. Hence, every instance of a class is mapped to an Entity. Whereas, each attribute in the classes has been mapped to an Entity Property. Ratings and questions attributes' data types are Lists of integers and strings respectively, and since List data type is natively supported by the Datastore it is automatically interpreted and no further action is necessary.
CPIM library does not make use of ancestor paths, hence data are stored so as that future operations will retrieve those data in an eventually consistent way.

Azure Tables The classes in Figure 5.1 are translated into two different tables by Azure Tables. For every class instance a new Entity (row) inside the respective table is created. Each class attribute is then mapped to an Entity Property.
Since ratings and questions attributes data types are Lists of integers and strings respectively, and they are not supported by Azure Tables, the respective Properties data type is set as binary array. To preserve original

data type information, CPIM library creates a new Property whose name is `jpa4azureBeanType`, which contains the name of the object (UserRatings or Topic) that will be used to deserialize the attributes labelled with the annotation `@Embedded` – i.e. ratings and questions.
CPIM library sets the same Partition Key for every entity stored inside the same table, this forces Azure Tables to store data so as that future operations will retrieve those data in a strongly consistent way.

5.2.1 Data migration: compatibility test

The first test, that has been conducted, aimed at verifying whether MiC application would have worked after complete data switch over from a NoSQL database to another.
In order to do so, MiC application has been deployed on Google App Engine and some data has been generated by simulating users answers. In particular 1,200,000 UserRatings Entities have been generated performing the following steps:

- email field has been generated starting from the email address `modaclouds<number>@polimi.it` and substituting `<number>` with a progressive number.
- topicName field has been randomly chosen among elements of a set composed by 7 different topics.
- ratings field has been generated by randomly choosing an answer among five of them.
- id field is simply the concatenation of the previous email field and topicName field.

Finally, these generated entities have been directly inserted in GAE Datastore by means of RemoteApi.
After that, the migration system, deployed on a local working station, has been executed and the task of migrating data from GAE Datastore to Azure Tables has been assigned to it. Details about migration system deployment and execution are reported in Appendix B.
At data migration completion, MiC application has been deployed on Windows

Azure. Almost all MiC functionalities have been tested and all of them worked as expected.
The only exception is related to the function that calculates users similarities. Similarity functions extract data contained inside ratings field and calculate Pearson's coefficient.
Ratings field contains a list of objects. Since lists are not natively supported by Azure Tables, CPIM library serializes ratings list and stores the resulting bytes as a Property, of byte array type, in Azure Tables. In the migration from GAE Datastore to Azure Tables this does not represent a problem, since CPIM uses JPA serialization algorithm, which is the same used by the migration system. Hence, serialized migrated data is correctly interpreted by MiC application.
Although this problem does not occur in the migration from GAE Datastore to Azure Tables, it arises after the inverse migration, from Azure Tables to GAE Datastore, has been done. In fact, since GAE Datastore natively supports lists, MiC application expects to read an object list from ratings Property, in order to apply similarity algorithms. But, during the inverse migration, the migration system reads a byte array from Azure Tables and migrates it to a byte array on GAE Datastore. This implies that MiC function that calculates users similarities does not work after this migration, because it expects to retrieve a list and not a byte array.
So, in this particular case, MiC application should be modified in order to read the original data type from the type Property, created by the migration system, and then deserialize data into a list.

5.2.2 Performance tests

This section describes the migration tests conducted over data stored by MiC application, both in GAE Datastore and in Azure Tables, from a quantitative perspective. In particular, the migration is performed on UserRatings class, since it contains more instances than Topic class. Furthermore, UserRatings class contains attributes of several data types, including collections – i.e. a list of integers – and, for testing purposes, other two boolean attributes have been added.
MiC uses different consistency policies on the two databases, i.e. data present in GAE Datastore have been stored in order to provide eventual consistency,

whereas, data in Azure Tables have been set up to preserve strong consistency on all data contained by the table UserRatings.
For this latter reason, we made several tests with different consistency policies for each database:

1. A migration from GAE Datastore to Azure Tables, where MiC source data is stored, by MiC application itself, in order to provide **eventual** consistency.

2. A migration form from Azure Tables to GAE Datastore, where MiC source data is stored, by MiC application itself, in order to provide **strong** consistency.

3. A migration form from Azure Tables to GAE Datastore, where data generated by the test at point 1 have been stored by the migration system, in order to provide **eventual** consistency.

Each of these tests has been performed into two different scenarios:

1. On a Virtualized Private Server located in Politecnico di Milano.

2. On the Google Cloud IaaS platform, i.e. Compute Engine.

Each test considers four aspects:

1. The throughput of data transiting in the system, i.e. entering entities, Metamodel entities in the Queue and exiting entities, as described in Chapter 3.

2. CPU usage.

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. conversion and extraction time.

The tools used to conduct the tests are:

- a linux virtual machine, whose characteristics depend on the scenario considered, hence each of the following paragraphs will make them explicit.

- sysstat package to measure the percentage of CPU used durings 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 customized library to measure the size of objects stored in the queue.
- Google App Engine statistics on Datastore to calculate the average dimension of entities stored and the total amount of data to be migrated.

In the following two paragraphs we provide the results for the two scenarios. In Section 5.5 we discuss the obtained results.

In House scenario

All tests have been performed on an Intel Nehalem dual socket quad-core CPUs @2.4 GHz with 24 GB of RAM running Ubuntu Linux 2011.4. The migration system runs inside a Tomcat application server, installed on a virtual machine with 3 dedicated physical cores and 8GB of RAM. The operating system installed on the virtual machine is Ubuntu Server 12.04 and it is configured as described in Appendix B.
Google does not provide any information about the physical location of the Datastore servers that host the MiC data. Whereas, data stored by Azure Tables is located in Western-Europe datacenters.
Figure 5.2 shows the deployment architecture for the conducted tests.

Google App Engine Datastore to Azure Tables migration For the migration from GAE Datastore to Azure Tables five tests have been conducted, each with a different number of entities to be migrated.
Based on Google App Engine statistics, an entity average size, for UserRatings Kind, is 454 Byte; hence, a proper number of entities has been chosen in order to transfer 16MB, 32MB, 64MB, 128MB, 256MB, and 512MB of data.
Given that, MiC does not use ancestor paths for storing data in GAE Datastore, every entity is a root entity; hence, data get translated to Metamodel representation preserving eventual consistency. So, based on the translation approach described in Chapter 4, we can conclude that the system performs a read operation from GAE Datastore and a write operation to Azure Tables

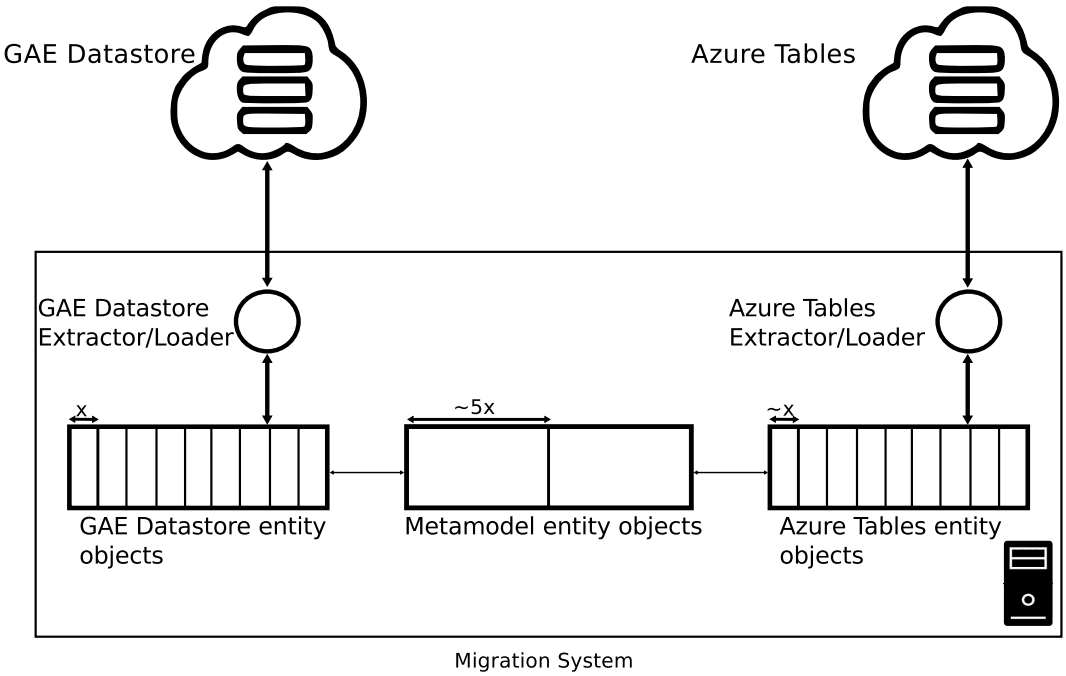

Figure 5.2: Deployment architecture - In House scenario

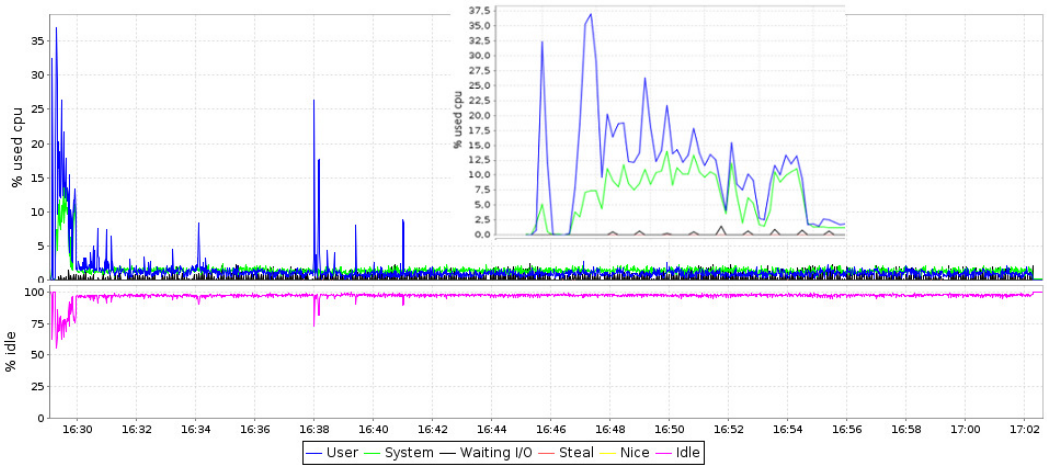

Figure 5.3: Migration from GAE Datastore to Azure Tables - CPU usage

Table 5.1: Migration from GAE Datastore to Azure Tables - In House scenario

	dataset #1	**dataset #2**	**dataset #3**	**dataset #4**	**dataset #5**	**dataset #6**
Source size	16	32	64	128	256	512
# of Entities	36940	73879	147758	295515	591040	1182062
Migration time (sec)	2009	3914	7915	16348	33156	61204†
Entities throughput (ent/s)	18.387	18.876	18.668	18.077	17.826	19.313
Queued data (MB)	81.98	166.90	336.73	676.00	1331.20	2709.00
Extraction and Conversion time (sec)	42	69	182	273	625	862
Queued data throughput (KB/s)	1998.75	2476.89	1894.57	2535.62	2181.04	3218.12
Exiting data (MB)	70.95	141.89	283.79	567.57	1135.16	2270.28
Exiting data throughput (KB/s)	36.16	37.12	36.72	35.55	35.06	37.98
Avg. %CPU usage	2.975	2.616	2.437	1.789	1.795	1.803

Tests started on Sunday afternoon at 6PM and ended on the next Tuesday at 5AM
†This particular test started on Thursday around 9PM

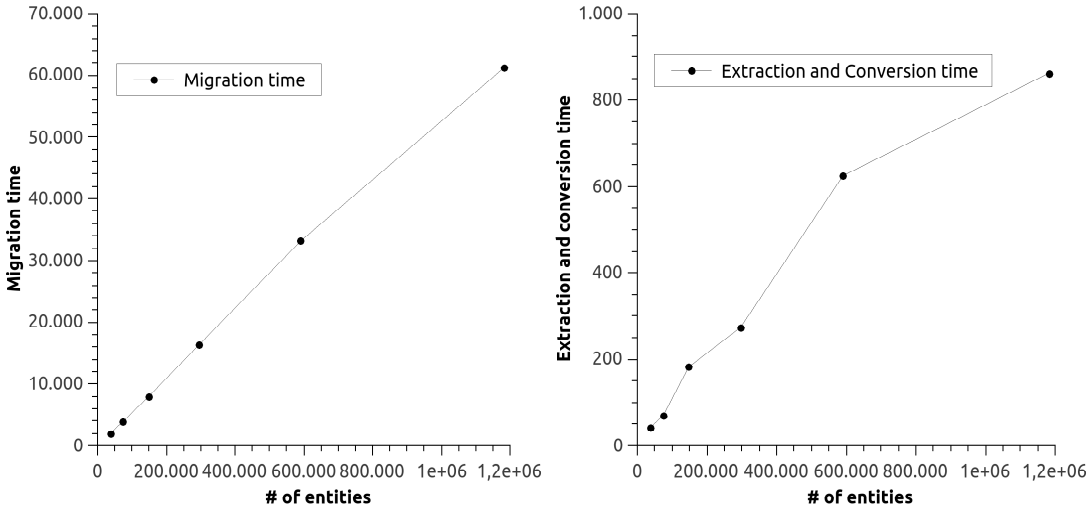

Figure 5.4: Migration from GAE Datastore to Azure Tables - Times growth - In House scenario

for each entity that is migrated.
Results of each test are reported in Table 5.1. Whereas, Figure 5.3 illustrates CPU usage variation during the migration; in particular, it shows that after Metamodel entities production, CPU usage drops to values between 2% and 3%, and remains almost constant for the whole migration procedure. During Metamodel entities production, the CPU experiences a greater usage, with values ranging between 37% and 15%.
Finally, Figure 5.4 depicts how migration times and extraction and conversion times grow, with respect to the number of entities.

Table 5.2: Migration from Azure Tables to GAE Datastore preserving eventual consistency - In House scenario

	dataset #1	**dataset #2**	**dataset #3**	**dataset #4**	**dataset #5**
# of Entities	9235	18470	36940	55410	73879
Migration time (sec)	4944	10141	15738	30804	29666
Entities throughput (ent/s)	1.868	1.821	2.347	1.799	2.490
Queued data (MB)	23.32	45.26	94.85	144.42	193.98
Extraction and Conversion time (sec)	13	24	42	58	69
Queued data throughput (KB/s)	1836.623	1930.953	2312.609	2549.672	2878.701
Exiting data (MB)	4.00	8.00	16.00	24.00	32.00
Exiting data throughput (KB/s)	0.83	0.81	1.04	0.80	1.10
Avg. %CPU usage	0.353	0.287	0.318	0.245	0.304

Tests started on Friday at 12AM and ended on the next Monday at midnight.

Azure Tables to Google App Engine Datastore migration preserving eventual consistency In this paragraph we report the data we have obtained migrating data that have been stored by tests performed in the previous paragraph, preserving eventual consistency. This caused every single entity, inside UserRatings table, to have a different Partition Key.
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, have been done, based on GAE Datastore entities.

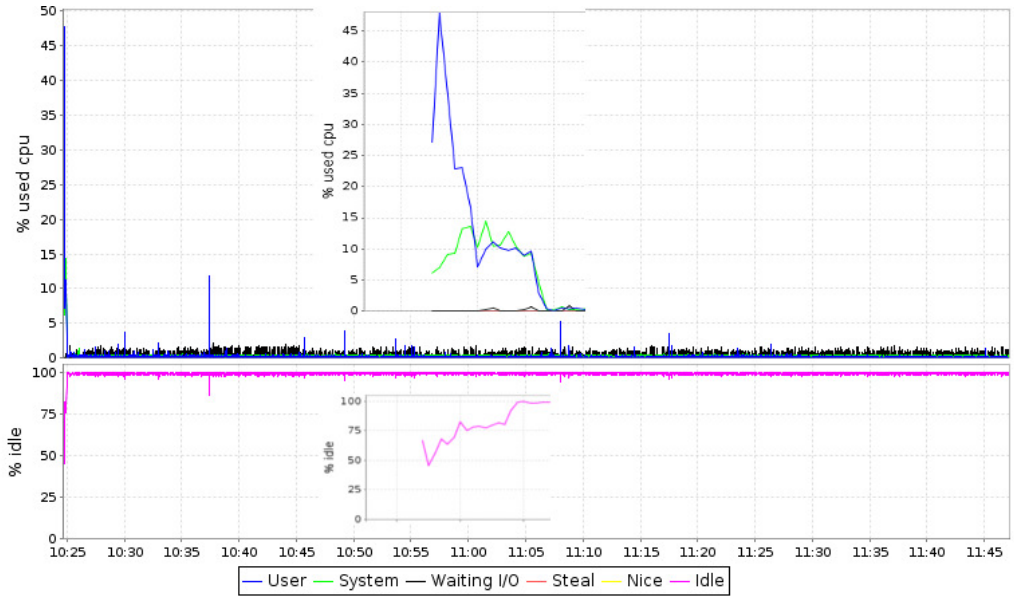

Figure 5.5: Migration from Azure Tables to GAE Datastore preserving eventual consistency - CPU usage

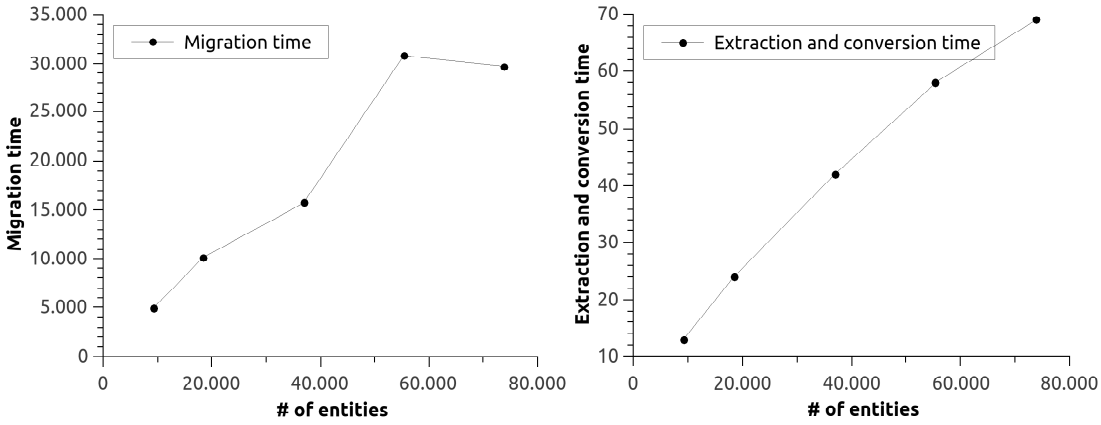

Figure 5.6: Migration from Azure Tables to GAE Datastore preserving eventual consistency - Times growth - In House scenario

During tests with more 70000 entities, we experienced frequent unrecoverable crashes – i.e. HTTP 500 and 503 errors – not depending on the migration system. These two latter errors refer to Google App Engine internal errors, i.e. GAE instances are not able to manage continuous HTTP requests, lasting more than a given amount of time. Hence, new approaches for transferring a bigger amount of entities should be implemented in the future. A possible solution is reported in Section 5.5.
In order to test the migration system in this situation, it has been decided to migrate a lower number of entities.
The migration system, after having mapped source entities to the Metamodel representation, checks their consistency policy and performs the proper translation to the destination database. Hence, based on the translation approach described in Chapter 4 and on the previous consideration on the Partition Keys, we can conclude that the system performs a read operation from Azure Tables, another read operation from GAE Datastore and two write operations to GAE Datastore, for each migrated entity.

Figure 5.5 illustrates CPU consumption during the migration. After Metamodel entities production, CPU usage drops to values between 0.2% and 1%, and remains almost constant for the whole migration procedure.
Finally, Figure 5.6 depicts how migration times and extraction and conversion times grow, with respect to the number of entities. From this figure we notice that the migration time does not seem to grow linearly; in particular we infer this from the last two tests – i.e. the one which transferred 55,410 entities and the one which transferred 73,879. We assume this behaviour is due to the multitenancy of the cloud database service, since the former test was executed on a Monday at 3PM (possible higher load), while the latter was executed on a Saturday at 16PM (possible lower load).

Azure Tables to Google App Engine Datastore migration preserving strong consistency This tests migrates MiC original data, stored with strong consistency policies, to GAE Datastore, thus preserving strong consistency. In fact, MiC assigns the same Partition Key to all entities contained by the same table. Hence, for every entity to be migrated, the migration system performs a read operation from Azure Tables, another read operation from

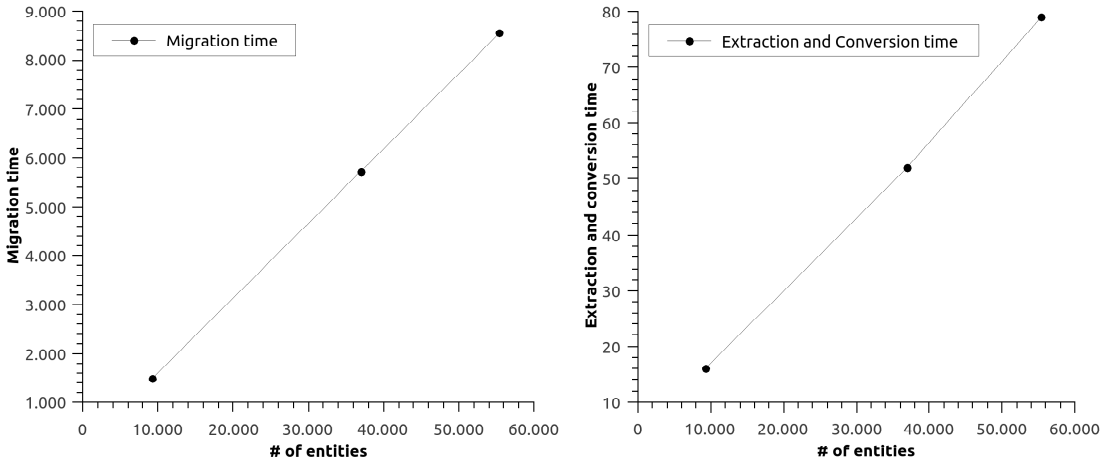

Figure 5.7: Migration from Azure Tables to GAE Datastore preserving strong consistency - Times growth - In House scenario

GAE Datastore – to check whether a parent entity already exists – and, since the parent entity has already been created by the first migrated entity, just one write operation on GAE Datastore is needed.
Details about this test are reported in Table 5.3.

Table 5.3: Migration from Azure Tables to GAE Datastore preserving strong consistency - In House scenario

	dataset #1	**dataset #2**	**dataset #3**
# of Entities	9235	36940	55410
Migration time (sec)	1487	5719	8547
Entities throughput (ent/s)	6.210	6.459	6.483
Queued data (MB)	22.76	91.02	136.53
Extraction and Conversion time (sec)	16	52	79
Queued data throughput (KB/s)	1456.339	1792.341	1769.653
Exiting data (MB)	4.00	16.00	24.00
Exiting data throughput (KB/s)	2.75	2.86	2.88
Avg. %CPU usage	0.830	0.755	0.690

Tests started on Thursday at 11AM and ended on the same day at about 11PM.

With respect to the previous test – i.e. where we preserved eventual consistency – we can observe:

- The overall throughput for transferring all the entities, preserving strong consistency, is almost three times greater.
- Extraction and conversion times are almost identical.

- The greater throughput seems to come at the cost of a higher average CPU utilization. Anyway the difference is negligible, and in both cases it settles below 1%.

Finally, Figure 5.7 depicts how migration times and extraction and conversion times grow, with respect to the number of entities

Cloud scenario

Since the migration system will probably be executed inside cloud environments, some tests on Google Compute Engine Cloud IaaS platform have been carried out.
Compute Engine is an IaaS platform and it allows to create virtual machines with different characteristics. For these tests, a virtual machine with two virtual CPUs and 7.5GB of RAM has been chosen. The virtual machine runs a Linux-based (Debian) operating system, and the test environment has been configured according to the previous Section 5.2.2. The virtual machine is physically hosted by Google datacenters in Western-Europe.
The same tests of the previous Section have been conducted, this time with just three different data sets, instead of six. The results are reported in the following paragraphs. Figure 5.8 shows the deployment architecture for the conducted tests.

Google App Engine Datastore to Azure Tables migration For the migration from GAE Datastore to Azure Tables three data sets, with different size, have been chosen: 16MB, 64MB and 512MB.
The results are reported in Table 5.4.

With respect to the In House scenario for the same test (reported in Section 5.2.2) we observed:

- Migration times reduced by an average factor of 1.82, with respect to the In House scenario.
- As a consequence of the previous point, the overall execution time results are 1.82 times greater, on average, with respect to the In House scenario.
- Extraction and conversion times are in general lower, of a factor 1.33, with respect to the In House scenario.

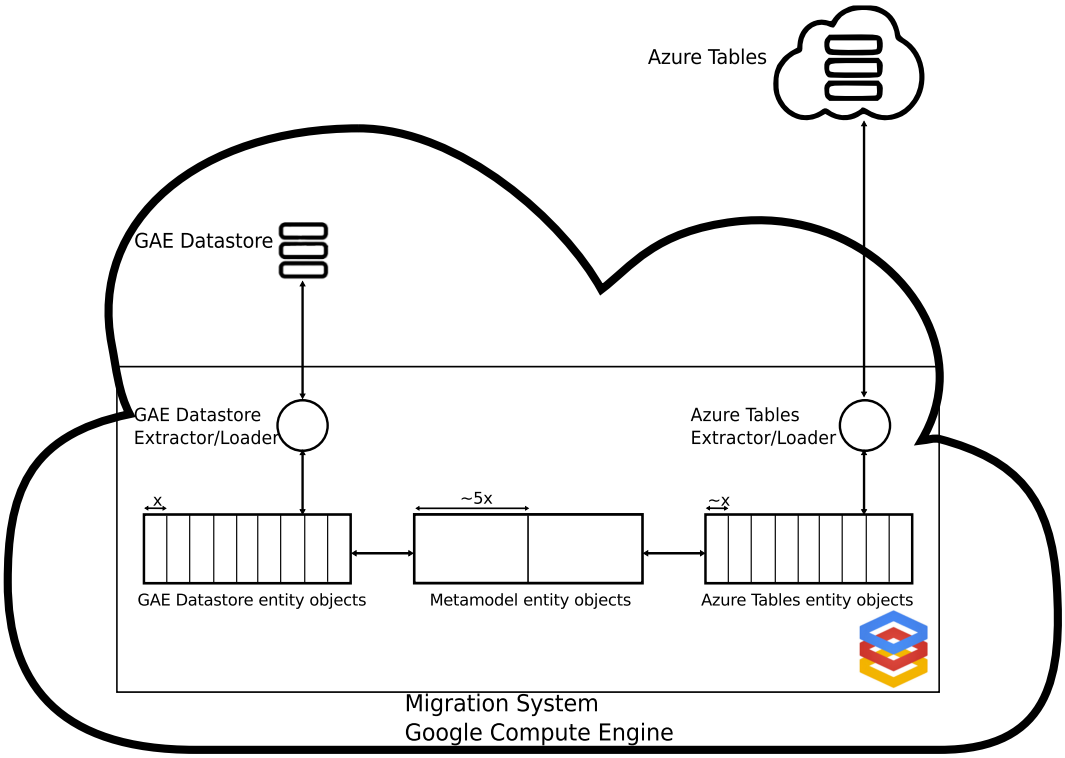

Figure 5.8: Deployment architecture - Cloud scenario

Table 5.4: Migration from GAE Datastore to Azure Tables - Cloud scenario

	dataset #1	**dataset #2**	**dataset #3**
Source size	16	64	512
# of Entities	36940	147758	1182062
Migration time (sec)	1098	4270	34111
Entities throughput (ent/s)	33.643	34.604	34.653
Queued data (MB)	81.98	336.73	2709.80
Extraction and Conversion time (sec)	31	120	768
Queued data throughput (KB/s)	2707.985	2873.446	3613.067
Exiting data (MB)	69.00	283.79	2270.28
Exiting data throughput (KB/s)	64.35	68.06	68.15
Avg. %CPU usage	4.749	3.947	4.111

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

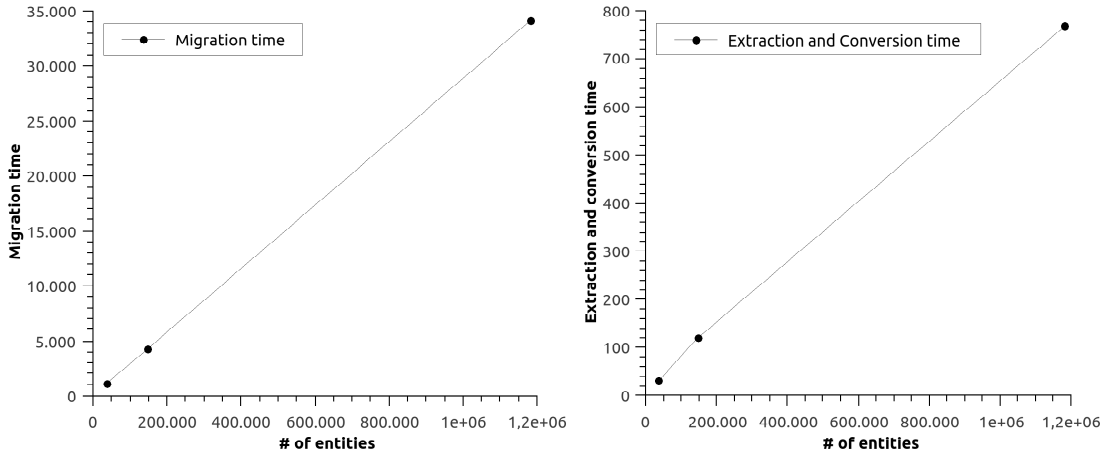

Figure 5.9: Migration from GAE Datastore to Azure Tables - Times growth - Cloud scenario

- Average CPU usage is in general greater, around values of 4%, with respect to the In House scenario, where it settles around values of 2%.
- As in the In House scenario, both migration time and the extraction and conversion time grow almost linearly with respect to the number of transferred entities.

Finally, Figure 5.9 depicts how migration times and extraction and conversion times grow, with respect to the number of entities.

Azure Tables to Google App Engine Datastore migration preserving eventual consistency This tests migrates data, generated from previous test, back to GAE Datastore preserving eventual 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, have been done, based on GAE Datastore entities. The three data sets considered, together with the results of the tests are reported in Table 5.5.

With respect to the In House scenario for the same test (reported in Section 5.2.2) we observed:

- Almost equal migration times, just a slightly lower than in the In House scenario. As a consequence the destination data throughput of both scenarios are almost equal.

Table 5.5: Migration from Azure Tables to GAE Datastore preserving eventual consistency - Cloud scenario

	dataset #1	**dataset #2**	**dataset #3**
# of Entities	9235	36940	73879
Migration time (sec)	4128	13101	27970
Entities throughput (ent/s)	2.237	2.820	2.641
Queued data (MB)	22.97	93.50	191.22
Extraction and Conversion time (sec)	10	24	44
Queued data throughput (KB/s)	2352.302	3989.513	4450.118
Exiting data (MB)	4.00	16.00	32.00
Exiting data throughput (KB/s)	0.99	1.25	1.17
Avg. %CPU usage	0.701	0.605	0.563

Tests started on a Tuesday at 9AM and ended on the same day at about 10PM

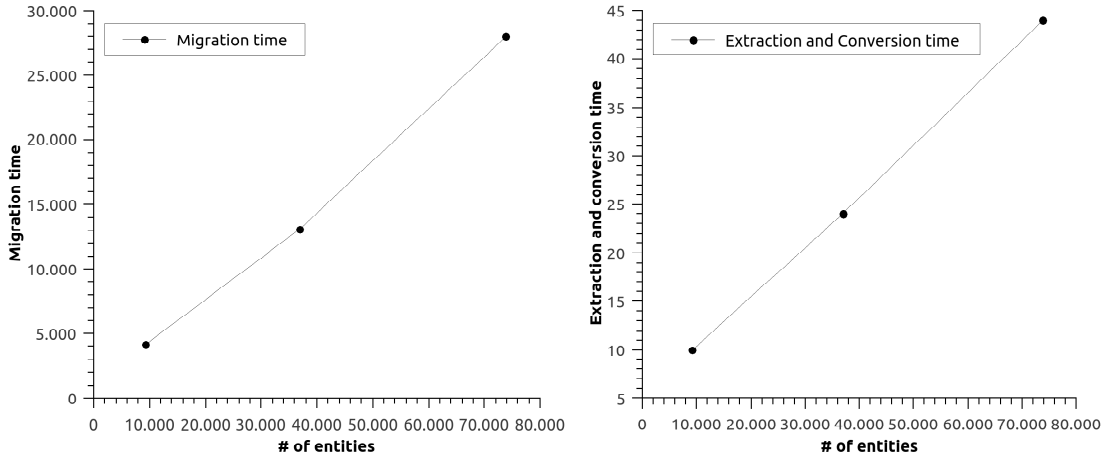

Figure 5.10: Migration from Azure Tables to GAE Datastore preserving eventual consistency - Times growth - Cloud scenario

- Extraction and conversion time is lower in this cloud scenario. This may be due to the variability of the cloud performance in general.
- CPU usage is below 1% in both scenarios.

Finally, Figure 5.10 depicts how migration times and extraction and conversion times grow, with respect to the number of entities.

Azure Tables to Google App Engine Datastore migration preserving strong consistency This test migrates MiC original data, stored in a strongly consistent way by MiC application itself, to GAE Datastore. The migration system will preserve strong consistency on the destination database, i.e. GAE Datastore.
The three data sets considered, together with the results of the tests are reported in Table 5.6.

Table 5.6: Migration from Azure Tables to GAE Datastore preserving strong consistency - Cloud scenario

	dataset #1	**dataset #2**	**dataset #3**
# of Entities	9235	36940	55410
Migration time (sec)	1402	5340	8599
Entities throughput (ent/s)	6.587	6.918	6.444
Queued data (MB)	22.40	89.61	134.41
Extraction and Conversion time (sec)	10	30	41
Queued data throughput (KB/s)	2294.067	3058.627	3357.047
Exiting data (MB)	4.00	16.00	24.00
Exiting data throughput (KB/s)	2.92	3.07	2.86
Avg. %CPU usage	1.509	1.139	0.957

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

With respect to the In House scenario for the same test (reported in Section 5.2.2) we observed:

- Almost equal migration times. As a consequence, the output throughput is almost identical.
- Extraction and conversion times are lower, in all tests, with respect to the In House scenario. Hence, the throughput on the queue is greater in this scenario.

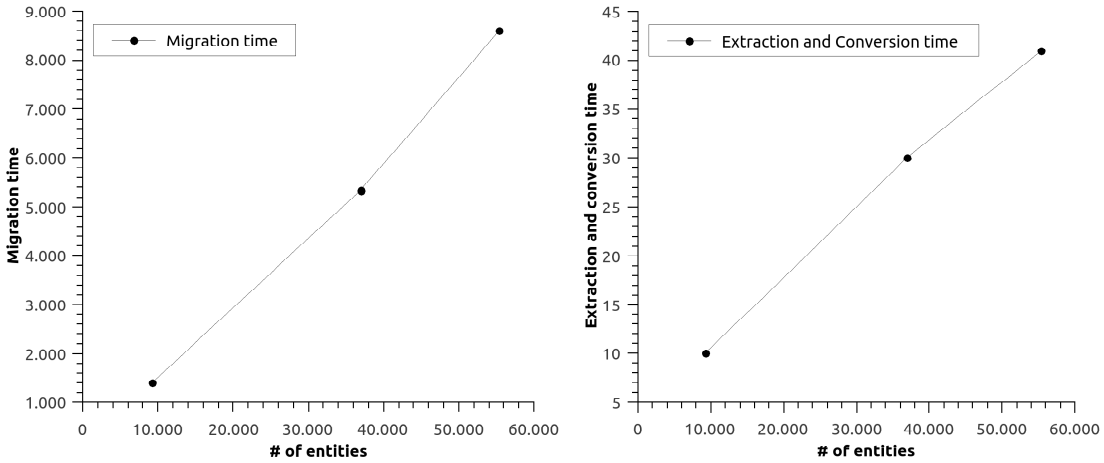

Figure 5.11: Migration from Azure Tables to GAE Datastore preserving strong consistency - Times growth - Cloud scenario

Furthermore, with respect to the previous test – i.e. where we preserved eventual consistency – we can observe:

- The overall throughput for transferring all the entities, preserving strong consistency, is almost three times greater.
- Extraction and conversion times are almost identical.
- The greater throughput seems to come at the cost of a higher average CPU utilization. Anyway the difference is negligible.

Finally, Figure 5.11 depicts how migration times and extraction and conversion times grow, with respect to the number of entities.

Table 5.7: GAE Datastore throughput test

	dataset #1	**dataset #2**	**dataset #3**
# of Entities	9235	18470	36940
Entities size (MB)	4	8	16
Populating time (sec)	16	22	55
Overall time (sec)	153	309	651
queue population throughput (ent/s)	577.188	839.545	671.636
Overall throughput (ent/s)	60.359	59.773	56.743
Avg. %CPU usage	2.975	2.319	1.750

Tests started on a Monday from 12AM

Throughput test In order to measure the optimal entities throughput of GAE Datastore, and then compare it to the throughput obtained with the

migration system, a custom application has been developed. This application extracts entities from GAE Datastore, changes their Kind – i.e. from UserRatings to Throughput – and put them in a queue. A consumer thread extracts modified entities from the queue and puts them back to GAE Datastore, until the queue is empty (data have been replicated).
Variables measured by this test are the overall completion time and the time needed to populate the queue. The test has been performed with three different entities data sets, respectively of 4 MB, 8 MB and 16 MB. Starting from these data, the respective throughput has been calculated.
The custom application has been written in Java and it has been run on Google Compute Engine platform with the same configuration of previous sections.
The results of these tests are shown in Table 5.7.
We can observe that the overall migration time grows linearly with the number of entities to be transferred. The population time is variable and it depends on the reading latencies from GAE Datastore. Finally, the overall throughput decreases, in an inverse proportional way, with respect to the number of transferred entities.

5.3 Comparing the proposed migration approach with direct database-to-database translations

In this section we make some considerations about the Metamodel proposed in this thesis, in contrast with direct database-to-database translators that could be developed to accomplish the same task.
A database-to-database translator is a custom software that maps data from a source database to a destination one. Every time a new database needs to be supported, two new translators need to be written, for every database already supported. Thus, if N is the number of databases then $N \cdot (N-1)$ translators should be developed. Hence, it is evident that the numbers of translators grows quadratically with the number of databases.
Furthermore, when a new database needs to be supported, the developer has to be familiar with all the programming interfaces and the possible issues specific for every database already supported.
With the development of a metamodel, instead, the number of translators that

needs to be developed grows linearly, in fact it is $2 \cdot N$. Developers does not need to know about details of the other supported databases, in order to write proper mappings that work with all other databases.
One of the advantage of direct translators could be performance; i.e. if the metamodel is not designed properly, it may heavily impact on the overall time needed for the migration, and it may also affect CPU usage.
From the the tests results of the previous section, it is evident that the mere translation process, from a source database to the Metamodel, requires always less then 0.1% of the total migration time. Moreover, the overhead on CPU, for the translation process, is always less then 50% and, after direct translation, it suddenly drops to values below 5%.
Therefore, we can assert that the overall performance are not affected by the introduction of this Metamodel.
This comparison does not take into account the inverse translation process, since even in the case of direct mapping between databases, at least a translation should be done. Furthermore, the major part of the whole migration time does not depend on translations, but on write latencies. The exactly same write latencies are experienced by any translator, whether it is a direct one, or it uses an intermediate metamodel.
Thus, to conclude, we can affirm that the introduction of this Metamodel does not affect performance, indeed it eases the developers' burden when writing translators, and it offers a clear interface to preserve properties like secondary indexes and different consistency policies.

5.4 Extendability of the migration system

As an added value of this thesis, we show how it is possible to support the migration even of Key-Value databases thanks to the Metamodel described in Chapter 3.
This section focuses on the migration of Amazon DynamoDB, a Key-Value NoSQL database with a data model similar to the Dynamo one, discussed in Chapter 2, but with a different underlying architecture.

DynamoDB data model

An Amazon DynamoDB database is made of tables, each table can contain several items and each item is a collection of attributes. Each attribute is a key-value pair, where the key is the attribute name, whereas the value stores the attribute value. Furthermore, an attribute can be single-valued or a multi-valued set.
Amazon DynamoDB is a schema-less database, hence the number of attributes, that an item can contain, is not fixed. The only mandatory attribute is the Primary Key. Amazon DynamoDB provides two different types of primary keys:

- Hash Type Primary Key: a single-valued hash attribute, used to create an unordered hash index.
- Hash and Range Type Primary Key: a double-valued attribute, whose first attribute is the hash attribute and the second one is the range attribute. An unordered hash index is built on the first attribute, whereas a sorted range index is built on the second attribute.

Finally, Amazon DynamoDB allows users to specify whether they want a read operation to be eventually consistent or strongly consistent.

Translators design

In order to design proper translators for Amazon DynamoDB, some considerations should be done.
Since DynamoDB provides two different consistency policies, even though just for read operations, a user expects that those policies are preserved by the migration system. Hence, two different pairs of translators can be designed, one for each type of consistency policy. Thus, upon data migration, the final user can choose the proper consistency policy for the data he is about to migrate.
The only difference between the two pairs of translators will reside in the respective direct translator, i.e. the one that translates source data to Metamodel data. In particular, in case strong consistency is chosen, the value of the Metamodel Partition Group field should be the same for any DynamoDB item contained by the same table. Whereas, in case of eventual consistency,

the Metamodel Partition Group value should vary with each DynamoDB item. Since, Metamodel Partition Group value is composed by two strings divided by an "#" symbol, in both cases the first string should coincide with the DynamoDB table name; whereas, the second string, in case of strong consistency, will be a fixed string for every entity ; or, in case of eventual consistency, it may coincide with the DynamoDB primary key .
The remaining mappings are the same for both of translators pairs.

Direct translator Each DynamoDB item will correspond to a new Metamodel Entity. Then, each Amazon DynamoDB table should be mapped to a Metamodel Column Family.
Based on the type of DynamoDB item Primary Key, two different actions should be performed:

- In case of Hash Type Primary Key, its value is directly mapped to Metamodel Entity Key.
- In case of Hash and Range Type Primary Key, it should be separated. The hash part is mapped to Metamodel Entity Key. Whereas, the second part is mapped to a new Metamodel Entity Column, automatically set to be indexable, in order to enable applications to express queries on its value.

Finally, each attribute contained by a DynamoDB item should be mapped to a Metamodel Entity Column. In case an attribute is indexable, a true value should be set in the respective Metamodel Column indexable field. Every attribute value should be serialized and mapped to a Metamodel Column Value, and the original data type should be stored inside Column Type field.

Inverse translator Since consistency in DynamoDB does not depend on the data model design, the Metamodel Partition Group can be ignored.
Each Metamodel entity can be translated into a DynamoDB item. Hence the Metamodel Column Family should be extracted from the Entity and mapped to a new DynamoDB table.
The entity key can be mapped to a DynamoDB Hash Type Primary Key. Notice that, since a Metamodel entity can contain Columns that, in turn, are contained by different Column Families, it may be necessary to duplicate the

same Entity key to different DynamoDB tables.
Each Metamodel Column, contained by an Entity, should be mapped to an attribute. This attribute should be set as indexable in case the respective Column indexable field value is true. Finally, Column value should be deserialized and mapped to an attribute value, if its data type is supported. If a check mechanism founds that a specific data type is not supported, Column value should be left serialized and mapped to the attribute value. Thus, a new DynamoDB attribute should be created, following the conventions, to host the original data type.

5.5 Discussion

Some considerations can be done, based on previous tests.

Performance

In almost all considered performance tests, the time needed to perform a complete migration grows linearly with the amount of data to be transferred – i.e. source entities. The only exception was noted when migrating data from Azure Tables to GAE Datastore preserving eventual consistency; we reckon this behaviour was caused by the multitenancy of the cloud database service, since one of the test was executed on a working day, while the other was started on a week-end day. Hence the database service may have served our requests under different workloads.
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 metadata that should be carried inside each Metamodel entity, in order to preserves transactionality, secondary indexes and data types, among the different databases.
From the analysis of CPU logs, it is evident that CPU spikes are caused by the translation of source data to the Metamodel representation. Nevertheless, CPU usage, during extraction and conversion phases, always settles below 50%. And, for the remaining migration time, CPU usage is almost constant around 2% value.
Moreover, extraction and conversion time occupies, on average, the 0.1% of the entire switch over time; hence, the overhead on CPU is negligible.
Another key factor is the difference on read and write latencies between the

two databases. The time needed to retrieve data from the source database – i.e. extraction time – is considered together with the conversion time, hence it is less then 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 with the number of entities retrieved from the source database. Hence, a solution to this problem may be given by the implementation of a queue that accepts only a limited amount of data; when the elements, not yet consumed, become lower than a certain threshold, the producer may retrieve more data from the source database. This approach would mitigate RAM consumption at the cost of a higher average CPU usage.
Tests conducted for preserving strong consistency on GAE Datastore required less time (3.22 times less) than those performed to preserve eventual consistency. This is due to the fewer number of write operations that the migration system needs to do, in order to preserve strong consistency.
The fact that test results, given in Section 5.2.2, assert that the overall migration time is almost two times less than those performed on Virtualized Private Server may be due to the different load levels of the respective database services, experienced during tests.
Moreover, having considered throughput results in Section 5.2.2, the migration from Azure Tables to GAE Datastore can be considered acceptable, especially for the fact that different consistency policies, as well as different data types, are preserved by the migration system.
Finally, as reported in Section 5.2.2, GAE Datastore frequently returned errors – i.e. HTTP error 500 and 503 – while trying to migrate more than 70,000 entities. This has led us to the conclusion that, as a future work, it will be necessary to divide the migration work – from the Metamodel to the destination database – thus, delegating a fewer number of entities to multiple threads. Each thread will handle a separate connection and will reliably migrate a smaller number of entities. The whole migration process will be considered finished when all threads will have persisted all entities assigned to them and when the central queue will be empty.

Database-to-database compared to metamodel approach

From performance tests and the analysis of the implications, due to the implementation of database-to-database translators, discussed in Section 5.3, we conclude that the proposed Metamodel does not have a significant overhead in the migration process. As a matter of fact, the usage of this Metamodel eases the developers' burden when writing translators to support a new database, since they do not need to have any further knowledge about already supported databases. Furthermore, the proposed Metamodel provides clear interfaces to preserve strong consistency and secondary indexes whenever possible.

Metamodel extendability

From Section 5.4 it is evident how the proposed Metamodel can support other types of NoSQL databases, like Key-Value ones. By performing correct mappings between the source database data model and the Metamodel (and viceversa), it is possible to migrate data among different classes of NoSQL database. Obviously, most of the times, secondary indexes, as well as strong consistency, would not be supported by Key-Value databases, because of the typically more restricted number of functionalities they support.

5.6 Summary

This chapter discussed several aspects about the migration system and the Metamodel proposed in Chapter 3.
A first test took into account the compatibility of migrated data with a real application. Other tests aimed at measuring performance of the migration system, in different scenarios and with different data sets. Then, several tests have been conducted in order to measure different properties of the migration system. Subsequently, some considerations about other possible alternatives to the metamodel approach have been discussed.
Finally, this chapter showed how it is possible to support also Key-Value databases, as well as Column-Based ones. By doing so, the chapter also described how it is possible to operate with databases that offer different consistency policies.

Chapter 6

Conclusions and Future Works

This thesis presented an original approach that enables the migration of data among different NoSQL databases, focusing in particular on Columnar NoSQL databases.

As reported in Chapter 2, about the analysis of the state of the art on NoSQL databases, 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 with these technologies becomes problematic.

Chapter 3 provides a solution to the above problem, by proposing an original 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 and secondary indexes.

The actual version of the migration system provides translators, as described in Chapter 4, for two databases: Google App Engine Datastore and Azure Table. Notwithstanding this, the premises are in place for additional support of other databases, as reported in Chapter 5.

The migration system, designed according to the latest Software Engineering techniques, 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 (described in Appendix A) which allows external applications to interact with it.

By using data generated by a real application, called MiC, some tests have been conducted, in order to evaluate system compatibility and performance. As reported in Chapter 5, performance tests have demonstrated that the overheads introduced by the migration system, in terms of latency and CPU

usage, are negligible in every circumstance.

Possible future works should investigate on the three main directives briefly discussed by the following three paragraphs:

Migration system generalization In order to support more databases, new mappings can be added to the migration system, following the guidelines reported in Chapter 4. This will also allow to perform parallel switch over on multiple destination databases.

Migration system functionalities extension Furthermore, by distributing translators on diverse virtual machines (which may communicate by means of message queues with the migration system), nearer to databases datacenters, it will be possible to reduce the workload and provide better latencies.
Another improvement would consist in providing continuous data migration, i.e. the possibility to perform the migration of new data as it gets inserted or updated. Which, together with the possibility to perform partial updates and automatic data consistency check among different databases, implies the adoption of versioning and lock mechanisms.

Paradigm shift Finally, a further step, towards the improvement of the migration system, would be the integration of the proposed metamodel with an abstraction layer, like CPIM, in order to offer a common interface to applications which make use of NoSQL databases. Such an integration would allow to implement applications which are independent from the underlying storage mechanism. Thus, in case one were to change database vendor, application data would easily be migrated and the application would continue to work without any re-engineering process.

Appendices

Appendix A

Rest API

A.1 Introduction

This Appendix describes REST APIs useful to interact with the migration system. The following sections describe the technology used to implement these APIs and the operations that can be executed through them.
Table A.2 reports a summary of all the operations supported and the respective object that should be returned.

A.2 Technology

REST is an architectural style, based on HTTP protocol. Everything that should be accessed by the APIs is a resource which is identified by an URI. Every resource should support the common HTTP operations. Furthermore, resources may have different representations, e.g. xml, json, plain text, html, etc. A client typically asks for one representation via HTTP.
The methods used in REST architectures are DELETE, GET, POST and PUT. The convention says that:

- DELETE method should be used to remove resources.
- GET method should read resources without side-effects, i.e. the resource should not be modified by the method.
- POST should be used to update a resource or create a new one.
- PUT method should be used to create a new resource.

Since the migration system has been developed using Java, Java Specification Request 311 (JSR) has been used in order to provide REST support to it. In particular, the specification is called JAX-RS and the implementation that has been used is Jersey.
Jersey permits to create a servlet that analyzes specific classes in order to identify RESTful resources. The servlet, upon receiving an HTTP requests, identifies the correct class and method to answer to the request.
Finally, in order to provide both JSON and XML representations for requested resources, Java Architecture for XML Binding (JAXB) has been used. JAXB simply maps a Java object, throughout reflection, to XML or JSON representation. Hence by setting a proper content-type to an HTTP request, it is possible to receive a response in JSON or XML.

A.3 API design

All API URLs listed in this appendix are relative to
`http://<server_address>/ModaCLOUD_DB/api`.
For example, the `api/credentials?name=datastore.username` API call is reachable at
`http://<server_address>:8080/ModaCLOUD_DB/api/credentials?name=datastore.username`
Table A.2 reports all the available API calls, together with the respective HTTP method to make the call. Furthermore, the type of the returned object is also indicated.

- The Status object indicates whether the request has been processed correctly or not. It is, also, always attached, as a separate object, to the response. It contains a mandatory field called `status` which can assume three different states: `OK`, `WARNING` or `ERROR`. Furthermore, the Status object may contain other two optional fields: `message` and `error_code`. The `message` field can be present independently from the state of the response. Whereas, the field `error_code` is returned only when the state of the response, contained by the `status` field, is `ERROR`.
  A list of the default errors, together with their respective error code is reported in Table A.1.

- The Property object is used to return the credentials stored, by the migration system, in the file `credentials.properties`. It contains two mandatory fields and a mandatory Status object. The first field is called `name` and it contains the name of the particular credential that has been requested. The `value` field contains the value of the particular credential that has been requested.

Two types of output formats are supported: XML(default) and JSON. To use any of them, simply change the Content-type in the HTTP request, i.e. set `application/xml` for XML format, or `application/json` for JSON format. Any non-200 HTTP response code, can be considered an error; the returned data will contain more detailed information.

Error code	**Message**
1	A connection is already in place. Disconnect first.
2	Unable to connect to the destination.
3	No active connection.
4	Unable to save credentials.
5	Credential not found.
6	Unable to delete credentials.
7	One or all of the selected databases are not supported.
8	Too few parameters. Check the documentation.

Table A.1: Default API errors

A.3.1 Switch Over API

To perform a complete data migration from a source database to a destination database, the following HTTP POST request should be made:
`api/switchover?source=<source_db>&destination=<destination_db>`
The values supported at the moment for source and destination databases are: `DATASTORE` and `TABLES`. Indicating that it is possible to request a complete data migration from Google App Engine Datastore to Azure Tables, and viceversa.
Switch over API are designed in such a way that it is possible to provide multiple destination databases. Hence, when this happens, data is simultaneously migrated to the indicated destination databases.
Once the switch over request has been issued, the migration process is started

Name	Description	HTTP Method	Result	Request
Switch over	Maps a database into others	POST	Status	api/switchover?source=DATASTORE &destination=TABLES&destination=...
Store credentials	Creates a new credential into the system	PUT	Status	api/credentials? name=datastore.username &value=polimi.modaclouds@gmail.com
Get credential	Gets the value for a given credential name	GET	Property	api/credentials? name=datastore.username
Update credential	Updates the value for a given credential name	POST	Status	api/credentials? name=datastore.username &value=polimi.modaclouds@gmail.com
Delete credential	Deletes a credential from the system	DELETE	Status	api/credentials? name=datastore.username

Table A.2: REST API calls

asynchronously by the migration system, and a Status object is returned to the caller. Proper actions can be taken by the caller based on the returned Status.

A.3.2 Databases credentials management API

The migration system allows a client to perform CRUD operations on the system credentials. To modify the credentials the following HTTP request should be made: `api/switchover`.
The parameters to be passed, and the HTTP method that should be used, depend on the type of request that should be issued by the client. A complete list of the requests that can be issued to modify the credentials, together with the respective HTTP method and parameters, is provided in Table A.2.
All of the requests, except for the one that reads credentials (GET), return only a status object. Whereas, the GET request returns a Property object, as described above.

Appendix B

Application usage manual

B.1 Introduction

This Appendix gives a brief overview of the system configuration and usage. Instructions are given for Linux operating system, based on Debian distribution; notwithstanding this, these guidelines can be easily applied to any other distribution and/or operating systems.

B.2 Environment configuration

The system works with any Java based application server. In this chapter we will provide configuration instructions for Tomcat application server. Furthermore, in order for the system to work properly, Java Development Kit 7 (jdk7) should be installed.
These premises can be accomplished by issuing the following command in the shell:

```
sudo apt-get install tomcat6 openjdk-7-jdk
```

After the installation has completed, proper folders should be linked and access permissions should be provided:

```
cd /usr/share/tomcat6
sudo rm -r webapps
sudo ln -s /var/lib/tomcat6/conf conf
sudo ln -s /var/lib/tomcat6/webapps webapps
sudo chmod 777 /usr/share/tomcat6/webapps
```

In order for Tomcat to be launched using jdk7, its configuration file should be modified. Hence the following line should be added to file `/etc/default/tomcat6`:

```
JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64
```

Furthermore, the text

```
JAVA_OPTS="-Djava.awt.headless=true -Xmx128m -XX:+UseConcMarkSweepGC"
```

should be substituted by

```
JAVA_OPTS="-Djava.awt.headless=true -Dfile.encoding=UTF-8
-server -Xms4096m -Xmx4096m -XX:NewSize=256m -XX:MaxNewSize=512m
-XX:PermSize=512m -XX:MaxPermSize=512m -XX:+DisableExplicitGC".
```

Values should be set according to machine RAM.

Finally, the `ModaCLOUD_DB.war` file can be copied to directory:

```
/usr/share/tomcat6/webapps
```

and Tomcat can be started:

```
sudo service tomcat6 start
```

Tomcat will take care of deploying the file and creating a folder named `ModaCLOUD_DB` that will contain the files extracted from the `ModaCLOUD_DB.war` file.

B.3 Application configuration

Before performing the switch over, proper credentials should be added to the `credentials.properties`.

There are two ways for doing this. The first consists in manual insertion of credentials in the credentials file, located in directory: `/usr/share/tomcat6/webapps/ModaCLOUD_DB/WEB-INF/classes/`.

Otherwise, the credentials API, described in Appendix A, can be used.

Fields that need to be set for configuring GAE Datastore are:

```
datastore.username=<email>
datastore.password=<email_password>
datastore.server=<server_address>
```

Whereas, for Azure Tables just one field, containing a hash string, is needed:

```
azure.storageConnectionString=<hash_string>
```

This string can be generated from Microsoft Azure portal, by accessing to the Tables service page and, thus, requesting an access key for the chosen storage account. In particular, the hash string should be:
`DefaultEndpointsProtocol\=http;AccountName\=<storage_name>;`
`AccountKey\=<access_key>`

Google App Engine Datastore configuration The migration system uses Google RemoteApi to interact with the Datastore, for this reason some lines of code need to be added to file `web.xml`, present in every application deployed on Google App Engine.

```
<servlet>
    <servlet-name>RemoteApi</servlet-name>
    <servlet-class>
     com.google.apphosting.utils.remoteapi.RemoteApiServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>RemoteApi</servlet-name>
    <url-pattern>/remote_api</url-pattern>
</servlet-mapping>
```

B.4 Application usage

In order to perform the switch over, an HTTP POST request, to the application server, should be made.
A return format (JSON or XML) should be chosen, hence the HTTP Content-Type field should be set to `application/json` or `application/xml` respectively.
The POST request to issue is the following:
`http://<server>:8080/ModaCloud_DB/api/switchover?`
`source=<source_db>&destination=<dest_db>`
Where typical values supported for `<source_db>` and `<dest_db>` are `DATASORE` and `TABLES`.
Once the request has been issued, the API will respond with a status indicating if the request has been accepted correctly. After that, the system will start to migrate data asynchronously to the selected destination database.

Bibliography

[1] Datastore: Types and property classes. `https://developers.google.com/appengine/docs/python/datastore/typesandpropertyclasses`. [Online; accessed 28-Oct-2013].

[2] Understanding the table service data model. `http://msdn.microsoft.com/en-us/library/windowsazure/dd179338.aspx`. [Online; accessed 28-Oct-2013].

[3] Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, and Giorgio Gianforme. A runtime approach to model-independent schema and data translation. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 275–286, New York, NY, USA, 2009. ACM.

[4] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: the sos platform. In *Proceedings of the 24th international conference on Advanced Information Systems Engineering*, CAiSE'12, pages 160–174, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[6] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali

Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[9] M. Shokrolahi Yancheshmeh D. Ardagna E. Di Nitto F. Giove, D. Longoni. An approach for the development of portable applications on paas clouds. Closer 2013 Proceedings, pages 591–601, Aachen Germany, 2013.

[10] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[11] Paolo Papotti and Riccardo Torlone. An approach to heterogeneous data translation based on xml conversion. In *CAiSE Workshops (1)*, pages 7–19, 2004.

[12] Ben Scoffield. Nosql – death to relational databases(?). Presentation at the CodeMash conference in Sandusky (Ohio), 2010-01-14., January 2010.

[13] Christof Strauch. Nosql databases. `http://www.christof-strauch.de/nosqldbs.pdf`, 2011. [online verfügbar: 16.09.2012].