# Cloud based NoSQL Data Migration Framework to Achieve Data Portability

Aryan Bansel

Submitted as part of the requirements for the degree
of MSc in Cloud Computing
at the School of Computing,
National College of Ireland
Dublin, Ireland.

September 2015

Supervisor Dr. Horacio Gonzalez-Velez

# Abstract

Cloud computing enables the user to access various services, such as infrastructure, platform, software, data storage with minimum efforts and cost. Adversely, the cloud providers introduce heterogeneity among these services, which in turn, makes the end users strictly dependent on specific solution. Though the NoSQL solution can manage large volumes of data as well as ensure high availability, scalability and fault-tolerance, it also provides several implementations since each cloud service storage requirements varies causing heterogeneity. Further, the end users or enterprises which design their application services using specific NoSQL based data model face difficulty in migrating the data according to business or technology changes. Vitally, the data portability enables the migration of data, and enhance interoperability across several cloud platforms. Therefore, the research aims at migration of data across cloud based heterogeneous NoSQL data stores since most of the cloud providers are adopting NoSQL solutions for scalability, and availability. As a proof of concept, the research presents a novel cloud based NoSQL data migration framework which enhances the data portability between different NoSQL implementations such as document, columnar, and graph. The approach involves data standardization and classification stages to ensure the data consistency and appropriate mapping. The research also compares the different methodologies and alternatives to evaluate the most viable performance-oriented approach to the data portability.

**Keywords:** Data portability, NoSQL, Key-value, Columnar, Document, Graph Database, NoSQL migration framework, extensibility.

# Submission of Thesis to Norma Smurfit Library,

# National College of Ireland

**Student name: Aryan Bansel Student number: 14104202**

**School: School of Computing Course: MSc in Cloud Computing**

**Degree to be awarded: MSc in Cloud Computing**

**Title of Thesis: Cloud based NoSQL Data Migration Framework to Achieve Data Portability**

One hard bound copy of your thesis will be lodged in the Norma Smurfit Library and will be available for consultation. The electronic copy will be accessible in TRAP (http://trap.ncirl.ie/), the National College of Ireland's Institutional Repository. In accordance with normal academic library practice all thesis lodged in the National College of Ireland Institutional Repository (TRAP) are made available on open access.

I agree to a hard bound copy of my thesis being available for consultation in the library. I also agree to an electronic copy of my thesis being made publicly available on the National College of Ireland's Institutional Repository TRAP.

Signature of Candidate:

For completion by the School:

The aforementioned thesis was received by___________________________ Date: _______________

This signed form must be appended to all hard bound and electronic copies of your thesis submitted to your school.

# Submission of Thesis and Dissertation

**National College of Ireland**

**Research Students Declaration Form**

**(Thesis/Author Declaration Form)**

**Name: Aryan Bansel**

**Student Number: 14104202**

**Degree for which thesis is submitted: MSc in Cloud Computing**

**Material submitted for award**

(a) I declare that the work has been composed by myself.

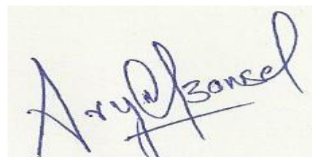
(b) I declare that all verbatim extracts contained in the thesis have been distinguished by quotation marks and the sources of information specifically acknowledged.

(c) My thesis will be included in electronic format in the College Institutional Repository TRAP (thesis reports and projects)

(d) **Either** *I declare that no material contained in the thesis has been used in any other submission for an academic award.

**Or** *I declare that the following material contained in the thesis formed part of a submission for the award of

**Master of Science in Cloud Computing awarded by QQI at level 9 on the National Framework of Qualifications.**

**Signature of research student:** 

**Date: September 19, 2015**

# Acknowledgement

An extensive endeavor, bliss euphoria that accompanies the successful completion of any task would not be complete without the expression of gratitude to the people who made it possible.

I take the opportunity to express my sincere gratitude to my project supervisor Dr. Horacio Gonzalez-Velez for providing me such opportunity. It would have never been possible for me to pursue the research upto this level without his relentless supervision and encouragement.

With an immense pleasure, I would like to thank Dr. Adriana Chis, Michael Bradford and Keith Brittle for their kind guidance and for providing all necessary information to accomplish the research. I highly appreciate the efforts and the timely advices they provided me to uphold motivation during studies.

I would like to extend my heartfelt thanks to my parents, family members and friends for their kind co-operation and moral support. My thanks and appreciations also go to my colleagues who helped in understanding the real world problems and all individuals who have willingly helped me out with their abilities.

# Declaration

I hereby declare that the dissertation entitled 'Cloud based NoSQL data migration framework to achieve data portability', is a bonafide record composed by myself, in partial fulfillment of the requirements for the MSc in Cloud Computing (2014/2015) programme.

This thesis is a presentation of research and development carried under the guidance and supervision of Dr. Horacio Gonzalez-Velez, Associate Professor and Head of Cloud Competency Centre.

Signed ........................ Date......................

Aryan Bansel

# Contents


**Abstract** **ii**

**Acknowledgement** **v**

**Declaration** **vi**

**1 Introduction** **1**

**2 Literature Review** **4**

2.1 Data Portability across Clouds . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 NoSQL based DataStores . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.1 NoSQL Common Characteristics . . . . . . . . . . . . . . . . . . . . 7
2.2.2 Key-Value Stores . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2.3 Document-oriented Database . . . . . . . . . . . . . . . . . . . . . . 9
2.2.4 Graph-based Database . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2.5 Column-based stores . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Data Migration across NoSQL DataStores . . . . . . . . . . . . . . . . . . 11
2.3.1 Heterogeneity in NoSQL Solutions . . . . . . . . . . . . . . . . . . . 12
2.4 Data Migration Stratgeies . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

**3 Design** **15**

3.1 Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2 Design Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Document Database to Graph-based Database . . . . . . . . . . . . . . . . 18
3.3.1 Data Transition to Graph . . . . . . . . . . . . . . . . . . . . . . . . 19
3.4 NoSQL Data Migration Framework . . . . . . . . . . . . . . . . . . . . . . 21
3.4.1 Document-based to Columnar Database . . . . . . . . . . . . . . . . 21
3.4.2 Data Translation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.4.3 Columnar NoSQL to Graph-based NoSQL . . . . . . . . . . . . . . . 24

**4 Implementation** **27**

4.1 Data Standardization . . . 27
4.1.1 Document Parsing . . . 30
4.1.2 Meta-modeling . . . 31
4.2 Data Classification and Staging . . . 33
4.2.1 Neo4j - Graph Implementation . . . 34
4.3 Direct Data Transformer . . . 38
4.3.1 Direct Mapping . . . 39

**5 Evaluation** **41**
5.1 Compatibility Tests . . . 42
5.1.1 MongoDB vs Azure Table vs Neo4j . . . 42
5.2 Cloud scenario . . . 43
5.3 Document to Columnar Database . . . 44
5.4 Columnar to Graph Database . . . 45
5.5 Document to Graph Database . . . 47
5.6 Query Performance . . . 49
5.7 In-house Scenario . . . 50
5.7.1 In-house Performance . . . 50

**6 Conclusion** **52**
6.1 Future Work . . . 53

**A NoSQL Data Migration Framework** **56**

# List of Figures


2.1 Comparison between NoSQL types . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Key Differences (Abramova & Bernardino (2013)) . . . . . . . . . . . . . 12

3.1 System Layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 Document to Graph-based Database . . . . . . . . . . . . . . . . . . . . 18
3.3 Data Modeling in Document and Graph-based Databases . . . . . . . . 19
3.4 System Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.5 Columnar Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.6 Data Translation Overview . . . . . . . . . . . . . . . . . . . . . . . . 24

4.1 System Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.2 MongoLab Connection . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.3 UML Diagram for Azure Migration . . . . . . . . . . . . . . . . . . . . 33
4.4 UML Diagram for generation of Graph Database . . . . . . . . . . . . . 35
4.5 UML Diagram for Direct Graph Generation . . . . . . . . . . . . . . . . 39

5.1 Deployment Architecture of NoSQL Migration Framework . . . . . . . . 43
5.2 Migration Time vs Source Size . . . . . . . . . . . . . . . . . . . . . . 45
5.3 Azure Table to Neo4j . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.4 MongoDB to Neo4j . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.5 Performance of Translations . . . . . . . . . . . . . . . . . . . . . . . 48
5.6 Throughput of Translations . . . . . . . . . . . . . . . . . . . . . . . . 48
5.7 Query Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.8 In-house Deployment Architecture . . . . . . . . . . . . . . . . . . . . 50
5.9 In-house Migration Performance . . . . . . . . . . . . . . . . . . . . . 51
5.10 In-house Migration Throughput . . . . . . . . . . . . . . . . . . . . . 51

A.1 MongoLab Database on Windows Azure . . . . . . . . . . . . . . . . . . 56
A.2 Real dataset (MongoDB Document) . . . . . . . . . . . . . . . . . . . . 57
A.3 Real data on Azure Table . . . . . . . . . . . . . . . . . . . . . . . . . 57
A.4 Real data on Neo4j . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

# List of Algorithms


1 Collection Classification . . . . . . . . . . 18
2 Document Classification . . . . . . . . . . 19
3 Translation to Graph . . . . . . . . . . 20
4 Translation to Columnar Database . . . . . . . . . . 24
5 Transforming to Graph Database . . . . . . . . . . 25

6 Mapping to Azure Table . . . . . . . . . . 32
7 Mapping MongoDB Values to Azure Table Rows . . . . . . . . . . 32
8 Transforming Azure Table to Neo4j Graph . . . . . . . . . . 36
9 Nodes and Relationships . . . . . . . . . . 37
10 Generating Graph . . . . . . . . . . 40

# Listings

4.1 Parsing Sub-documents in document . . . . . 30
4.2 Parsing Conditions . . . . . 31
4.3 Azure Connection . . . . . 34
4.4 Dynamic Entity Filter . . . . . 36
4.5 Data Staging . . . . . 37
4.6 Building Processing Stores . . . . . 38
4.7 Establishing relations and properties . . . . . 40

# List of Tables


3.1 Document Mapping with Graph . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2 Columnar and Graph Database Mapping . . . . . . . . . . . . . . . . . . 25

4.1 Direct Mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

5.1 Neo4j Memory Mapping Calculations . . . . . . . . . . . . . . . . . . . 42
5.2 MongoDB to Azure Table . . . . . . . . . . . . . . . . . . . . . . . . 44
5.3 Azure Table to Neo4j . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.4 MongoDB to Neo4j . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

# Chapter 1

# Introduction

With the evolution of cloud computing as a new paradigm, an ability to facilitate computing resources and utilities has been significantly enhanced. Cloud computing exhibits pay-as-you-go pricing schemes to support on demand access to services, namely, infrastructure, platform and software (IaaS, PaaS, and SaaS respectively). Importantly, the cloud computing also enables database-as-a-service (DaaS) to facilitate different data or storage models to end users.

Unfavorably, the use of different data models introduce the heterogeneity in cloud services which makes difficult for a user to migrate the software from one specific cloud based storage to another. Cloud based storage models are responsible for storing and processing large amount of data in a scalable manner. During an initial phase of software development stage, it is difficult for developers to anticipate the growth and storage requirements of application which in turn compells the developers to design the application modeling specific data service.

As a matter of fact, there are two paradigms for modeling the data storage based on cloud, namely, NoSQL and relational databases. Traditional database management systems (RDBMS) does not supports horizontal scaling as well as suffers from poor performance in a distributed environment. On the other hand, NoSQL (Not only SQL) eliminates the problems that traditional RDBMS exhibits, in fact, possesses the ability to process the large volumes of data.

Vitally, there are several implementations of NoSQL since different cloud vendors implements the different strategies for managing their data models. Indeed, there is a strong need to achieve data portability across cloud based different data stores without any considerable cost.

Subsequently, the research studies the cloud based different NoSQL data stores such as MongoDB, Azure Table, and Neo4j. In addition, the study may enable the consumer to switch the service provider if a user is dissatisfied with service or due to some business and technology changes.

The research problem- **'Is it possible to migrate the data across cloud based heterogeneous NoSQL data stores such as MongoDB, Azure Table and Neo4j using metamodel driven framework?'** -studies the data migration across cloud based different NoSQL based data stores, in particular, document, columnar and graph based databases. Significantly, the research claims that the metamodel driven framework shall provide a generic model to eliminate the heterogeneity among cloud based different NoSQL datastores.

To address the aim, the research is organized into four main phases, first, **Literature Review 2** highlights the relevance of data portability across NoSQL solutions as well as challenges which obstructs the adoption of data portability in cloud. Moreover, the chapter compares and contrasts the several supporting methodologies and strategies to eliminate those challenges.

Further, the second phase of the research, **Design 3** discusses the proposed metamodel driven framework to achieve data portability among cloud based NoSQL datastores. In brief, the study presents the system architecture which explains the working schema of NoSQL data migration framework. Additionally, the research attempts to design a few fundamental algorithms which shall support in establishing an efficient mapping between multiple source and target NoSQL datastores.

Given the pivotal role that metamodel driven framework plays in migrating the data using fundamental mapping, the research implements the multiple NoSQL solutions such as MongoDB, Neo4j, and Azure Table in **Implementation 4** phase. Moreover, the chapter provides a detailed modeling technique to develop the NoSQL data migration framework using data classification and translation algorithms.

Subsequently, in **Evaluation 5** phase, the research demonstrates the cloud based NoSQL migration framework using real dataset to determine the system behavior. Furthermore, the chapter compares the performance of each NoSQL database and efficiency of all the translations involved. Significantly, the study also presents the observations while configuring the application in-house and highlights the several approaches which shall support the extension of proposed framework.

Before proceeding to the next section, we must understand what points of interest, the research shall highlight:

(i) How data consistency within a data migration process can be managed to ensure data quality?

(ii) What are the key challenges in defining a meta format for different NoSQL databases?

(iii) Can data standardization and classification address the fundamental issues of metamodel driven NoSQL migration framework?

**Expected Contributions**

In an attempt to migrate the data across cloud based heterogeneous NoSQL data stores (document, columnar, and graph), the research may offer the following benefits:

- Data portability shall eliminate the heterogeneity among multiple NoSQL data models, thus, the research encourages a generic cloud based NoSQL migration framework.
- The research would enable the service users to switch the NoSQL solutions in case of dissatisfaction, better alternatives, or technology changes without any considerable cost and efforts.
- The proposed approach may eliminate inaccuracy, inconsistency while migrating the data to target data store, since the research employs data classification and standardization mechanisms.
- The research models an extensible system which can further be extended to support multiple types of NoSQL databases.

# Chapter 2

# Literature Review

To appreciate the novel idea, it is essential to understand the necessity of data portability and various strategies adopted to adddress the challenges which results in heterogeneity among cloud based data models. Consequently, the review discusses and compares the related approaches which aims to achieve data portability.

Subsequently, the review is classified into four sections, first, **Data Portability across Clouds 2.1** which discusses the need of data portability to address the heterogeneity in multiple cloud semantics resulting in vendor lock-in situation. Further, the review highlights the heterogeneity at different levels and how the data portability plays an important role in eliminating the vendor lock-in problem in detail.

After learning the importance of data portability, the second section, namely, **NoSQL based DataStores 2.2** explains why the research problem attempts to migrate the data across cloud based NoSQL datastores. To illustrate, the review compares the traditional relational database management system (RDBMS) with Not only SQL (NoSQL) paradigm. Also, the review describes different categories of NoSQL and evaluate the most viable one to address the research problem. Significantly, the research identifies the various constraints and factors such as consistency, availability and scalability while migrating the data from one cloud datastore to another.

Post to understanding the NoSQL paradigm, the research explains the data migration phenomenon and examines the feasibility of data migration across heterogeneous NoSQL datastores. The section **Data Migration across NoSQL DataStores 2.3** also discusses the differences in data format of multiple NoSQL solutions.

In the latter part of the review, different data migration strategies have been presented to determine the most viable approach to address the research problem. Importantly,

the **Data Migration Strategies 2.4** section attempts to model a novel approach which shall encourage to migrate the data across different NoSQL solutions.

## 2.1 Data Portability across Clouds

This part of the study explains the need for data portability across different cloud platforms such as Google App Engine (GAE), Microsoft Azure, and Amazon, etc.- prior to this, it is important to understand the data portability.

According to Shirazi, Kuan & Dolatabadi (2012), **cloud portability means an ability to move the data and application services from one cloud provider to another.** In other words, portability is an ability which challenges heterogeneity in semantics of different clouds or cloud based different data models and eliminates vendor lock-in problem. As a matter of fact, the research problem attempts to achieve the data portability across heterogeneous data models based on cloud, therefore, it may be advantageous to study how the vendor lock-in and semantics heterogeneity restricts the user.

In context, Scavuzzo, Di Nitto & Ceri (2014) state that the different cloud providers advocate the different strategies and methods in implementation which results in cloud heterogeneity. As a consequence, the users may not be able to move their services from one cloud provider to another without any significant cost. Further, Ranabahu & Sheth (2010) highlight the lock-in problem at two different levels:

- Vertical Heterogeneity: A heterogeneity in infrastructure and resources i.e. when different IaaS providers advocates different implementation policies and forms proprietary such as Amazon, GoGrid, etc.
- Horizontal Heterogeneity: A user is unable to gain control or switch the cloud platform service (PaaS) providers without efforts and cost.

Additionally, Thalheim & Wang (2013) state that in order to migrate the data, one need to have a thorough understanding of the data source which includes different aspects such as data availability, data constraints since different data sources are designed by different modelling strategies and semantics. Further, the data source may have inconsistent, duplicate or inaccurate data. On the other hand, the target systems may require to incorporate some additional constraints on migrated data in order to ensure consistency.

In addition, Sellami, Bhiri & Defude (2014) state that

> Cloud environment usually provides one data store for the deployed applications. However, in some situations, this data store model do not support the whole applications requirements. Subsequently, an application needs to migrate from one data store to another in order to find a more convenient data store to its requirements. (pp.654)

Consequently, the review focusses on easy data migration across different cloud service providers to address horizontal heterogeneity.Also, the data portability may enable the users to switch the service providers according to the service level agreement (SLA) benefits, cost reductions, and consistency policies.

Given the importance of data portability, it is essential to learn the different data models and study the most viable model adopted by multiple cloud providers.

## 2.2 NoSQL based DataStores

Post to understanding the need of data portability, this section compares and contrasts the different type of data models as well as the factors which make NoSQL more prominent than others.

According to Shirazi, Kuan & Dolatabadi (2012), in a cloud computing environment, each cloud provider models the database in different manner based on their strategies. These strategies result in heterogeneous semantics for modeling the cloud storage and makes difficult for consumers to switch. Further, Shirazi, Kuan & Dolatabadi (2012) state that there are usually two data modeling paradigms, namely, Not only SQL (NoSQL) and relational database. In context, Dharmasiri & Goonetillake (2013) state that the NoSQL system significantly differs from traditional relational database management system (RDBMS) since **NoSQL does not require fixed structures for tables unlike RDBMS**. However, NoSQL systems exhibit a vast heterogeneity as there are several numbers of implementations of NoSQL. Consequently, it is very challenging to switch from one system to another since NoSQL systems involve processing of large volumes of data.

On the other hand, Shirazi, Kuan & Dolatabadi (2012) specify that traditional RDBMS lacks a horizontal scaling ability as well as RDBMS does not fit well for the cloud architecture due to poor performance in a distributed system environment. In addition, Chen et al. (2012) mention that NoSQL system exhibits high availability, flexibility, and can manage huge amount of data, thereby, it results in high performance.

Since NoSQL ensures high availability and scalability, North (2010) states that most

of the high-volume websites and cloud computing based enterprises, applications such as eBay, Twitter, Amazon, Google and Facebook are adopting NoSQL based data stores. Further, North (2010) mentions that BigTable is developed by Google to support distributed data, thereby, a vast number of applications such as Google Earth, Gmail, and Google Maps use BigTable based on NoSQL. Similarly, Amazon uses DynamoDB, a key-value NoSQL data store for high fault tolerance, availability and efficient processing. Additionally, Scavuzzo, Di Nitto & Ceri (2014) specify that Microsoft Azure Table stores the structured data in key-value format without schemas, and is efficient in managing applications that store huge amount of non-relational data.

Subsequently, it would be beneficial to study the characteristics possessed by several NoSQL based cloud data stores.

### 2.2.1 NoSQL Common Characteristics

Whilst it can be seen NoSQL databases can process and store large amounts of data more efficiently than relational databases. The efficiency and good performance are gained by NoSQL based data stores by loosening the explicit and implicit constraints on consistency, scaling, and availability.

Further, Sattar, Lorenzen & Nallamaddi (2013) specify that since NoSQL has a distributed and fault-tolerant model as well as does not comply with RDBMS model, it does not guarantee ACID (atomicity, consistency, isolation, and durability). In context, Frank et al. (2014) explain the ACID properties in brief such as atomicity means all or nothing i.e. either transaction is fully completed or canceled before operation. Secondly, consistency must be ensured by examining whether the integrity constraints are obeyed or not. Futhermore, isolation ensures that the transactions are executed independently of each other in the same time. Finally, the durability evaluates that committed transactions should be persisted may be on a secondary storage.

As a matter of fact, NoSQL databases model eventual consistency, moreover, the transactions may be limited to datum in some databases. In contrast to ACID model, Tauro et al. (2013) state that NoSQL incorporates CAP (consistency, availability, and partition tolerance) theorem which describes the response of distributed systems on receiving write and read requests.

Given the importance of CAP theorem obeyed by NoSQL, it is essential to study the model, thus, Tauro et al. (2013) explain the CAP attributes as following:

**Consistency:** This property ensures that system reads the data written at last from the same node or different node. In other words, a client shall always view the

newly written or updated one but won't be able to see the older data.

**Availability:** A system must generate the reasonable response within a time duration, moreoever, the rule applies to both the operations i.e. read and write request. Significantly, availability ensures that system must remain available even when some of nodes are down.

**Partition Tolerance:** The partition tolerance in distributed system ensures that the nodes in a system mustn't communicate with each other. Further, a partition maybe a temporary loss of connectivity or just a packet loss, but when the network is partitioned- distributed system still continues to work.

However, many large scale applications and ecommerce platforms adopts BASE (Basically available, soft-state and eventual consistency) model in case availability and partition tolerance are more prominent than consistency.

Consequently, the review advocates the NoSQL based data stores for migration across different cloud platforms. Further, it is important to study the various NoSQL types which models CAP theorem, in context, Shirazi, Kuan & Dolatabadi (2012) illustrate that NoSQL can be categorized into four types:

**Key-value Stores :** A model based on keys-values which is easy to implement, but inefficient in updating, and querying the part of a value.

**Document-oriented databases:** In this, semi-structured documents are stored in JSON format. It supports efficient querying and manages nested values with associated keys.

**Column family databases:** An efficient model that stores and processes large amount of distributed data over multiple machines.

**Graph databases:** Graph database enables scalability across multiple machines and allows data-model specific queries.

Figure 2.1 describes the differences between multiple NoSQL implementations in terms of performance, scalability, and complexity.Indeed, majority of the NoSQL solutions possess high performance in comparison to RDBMS.

Since the research intends to migrate the data across heterogeneous NoSQL solutions based on cloud, it is important to understand the aforementioned NoSQL types in detail.

| | 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 |

Figure 2.1: Comparison between NoSQL types

### 2.2.2 Key-Value Stores

In key-value store, Chandra (2015) states that the data is converted into set of keys which are stored in the lookup table. Subsequently, the keys are transformed into the location of the data on requirement. According to McKnight (2014*a*), a key-value store shall become obsolete soon since a column store or a document store have more prominence in market. However, the key-value store could be used in case of simplicity and commonality.

Examples of key-value databases:

- MemcacheDB
- Riak
- BerkeleyDB

### 2.2.3 Document-oriented Database

According to Chandra (2015), in document-based NoSQL data store, data is stored as documents and these documents are called collections. Further, the document based database offers an advantage of storing the data of different types as well as porting. Additionally, McKnight (2014*a*) states that the document database are far better in logging online events even when the events have varying features.

Importantly, Chandra (2015) emphasize that document based store focusses only on documents storage and access optimization instead of rows and records. However, the document database performs slower than a RDBMS and sometimes require more space for storage.

Examples of document-oriented database (supports JSON, YAML, BSON, and XML format):

- MongoDB
- CouchDB
- MarkLogic

### 2.2.4 Graph-based Database

Chandra (2015) state that when an information is stored into multi-attribute tuples which exhibit the relationships in a graph form i.e. involving nodes and edges, the storage container is called graph data store (GD). Moreover, GDs are efficient in traversing the edges (relationships) between several entities.

Also, Chandra (2015) enumerates certain features of GD such as Neo4j, and OrientDB are used for location-based, social, biological networks and services:

- ACID compliant
- Date model: Nodes, relations, and key-value pairs on both,
- Incorporate Euler Graph theory.

Mostly, the graph-based databases are widely used in social networking sites to establish relations and properties between multiple elements as well as efficient in processing large amount of data.

### 2.2.5 Column-based stores

According to Chandra (2015), column-oriented data stores are used in building high-performance applications since it avoids I/O overhead as well as utilize the memory efficiently. In addition, McKnight (2014*a*) specify that column data stores are excellent for semi-structured data which exhibits commonality as well as differences. Further, Chandra (2015) state that columnar databases operate using column rather than row and does not comply with RDBMS rules.

A few examples of columnar stores are:

- Cassandra
- HBase
- Azure Table

After learning the types of NoSQL data stores, the review shall explain the phenomenon of data migration across these data stores based on cloud platforms such as Azure Table, MongoDB, Neo4j, and DynamoDB.

## 2.3 Data Migration across NoSQL DataStores

To appreciate the above inference, we must examine the data migration across NoSQL based cloud data stores (DaaS) such as Neo4j, Microsoft Azure's Tables, and MongoDB.

In context, Scavuzzo, Di Nitto & Ceri (2014) explain that the **data migration as a process of data transfer between multiple databases.** In addition, Scavuzzo, Di Nitto & Ceri (2014) mention that the data migration process must limit the user interaction and should be automatic. Further, the process does not only involve transfer of data from one datastore to another, but it is also required that data must adapt to the target's database format and structure. Subsequently, the phenomenon involves huge risk to the consistency and integrity of data.

Therefore, the review proposes a system such that it should be agnostic to data generation and uses libraries for updating and maintenance. Futhermore, to address the problem of consistency, the review attempts to design the system which is independent of the application hosted and perform efficiently at data level. In contrast, Atzeni et al. (2009) provide a model-independent platform for data translation and can manage object-oriented, relational, object-relational, and XML-based variants. Though Atzeni et al. (2009) proposes the common programming interface, but it adopts the XML in conjunction with SQL for modeling such system, thus, the system doesn't fit with cloud data store specifications.

Further, Shirazi, Kuan & Dolatabadi (2012) attempts to migrate the data from NoSQL column family to graph database using design patterns. Additionally, they suggest that the data portability between Neo4j and HBase can be achieved using design patterns. Since, the graph databases are good in managing the very huge amount of data, as well as column family databases can significantly store and process the large volumes of data.

But the document based data stores give poor performance while retrieving a record value since it traverse the whole document structure (collection) and update unnecessarily. Consequently, it may be advantageous to enable the users to migrate the data from one NoSQL data store to another on the basis of business requirements or technology changes.

Indeed, it is important to learn the differences between heterogeneous NoSQL solutions to generate the metamodel which shall define the common representation of the data.

### 2.3.1 Heterogeneity in NoSQL Solutions

Taking an above study into account, this section shall compare and contrast the three essential NoSQL solutions: Document, Columnar, and Graph in order to address the research problem.

As we have studied earlier, columnar NoSQL stores are more efficient in storing and processing the data in comparison to document-based NoSQL data stores. In context, Shirazi, Kuan & Dolatabadi (2012) state that the columnar NoSQL databases can handle the large amount of data easily as well as the big enterprises such as Google, Facebook and Twitter are adopting columnar NoSQL solution.

Likewise, Abramova & Bernardino (2013) provide the differences in performance of Document and Columnar NoSQL solutions under different workloads and data volumes. Further, Abramova & Bernardino (2013) demonstrate the major differences between Document-based NoSQL store (MongoDB) and Columnar NoSQL store (Cassandra).

| | MongoDB | Cassandra |
|---|---|---|
| Development language | C++ | Java |
| Storage Type | BSON files | Column |
| Protocol | TCP/IP | TCP/IP |
| Transactions | No | Local |
| Concurrency | Instant update | MVCC |
| Locks | Yes | Yes |
| Triggers | No | Yes |
| Replication | Master-Slave | Multi-Master |
| CAP theorem | Consistency, Partition tolerance | Partition tolerance, High Availability |

Figure 2.2: Key Differences (Abramova & Bernardino (2013))

To illustrate, Abramova & Bernardino (2013) highlight the key differences between MongoDB and Cassandra, i.e. MongoDB exhibits CP (Consistency and Partition tolerance) type system whereas Cassandra and HBase models PA (Partition tolerance and Availability).In contrast, columnar Azure Table model all three features, however, consistency is more prominent over availability. Further, Cassandra or most of the Columnar NoSQL stores uses peer-to-peer replication, namely, Multi-master whereas MongoDB and other Document-based databases practice Master-Slave replication.

In contrast to the use of document and columnar databases, McKnight (2014*b*) emphasize that the graph database is the only NoSQL solution which fits with modern

workload growth and is highly performance-oriented. Importantly, McKnight (2014*b*) state that

> Graph databases also yield very consistent execution times that are not dependent on the number of nodes in the graph, whereas relational database performance will decrease as the total size of the data set increases.

Further, Qi (2014) advocates that the graph database enables a fast traversal among nodes as well as graph databases can easily model complex relationships. As matter of fact, McKnight (2014*b*) and Qi (2014) specify that graph database possess multiple benefits as opposed to other NoSQL and SQL databases such as efficient query execution, scalability, and extensibility. Moreover, Qi (2014) state that the most of the graph-based databases supports ACID model as described earlier.

Indeed, Gudivada, Rao & Raghavan (2014) express that big enterprises including Google, Facebook, and Twitter adopts graph data model since graph database can describe static as well as dynamic relationships in comparison to other NoSQL solutions. Furthermore, Gudivada, Rao & Raghavan (2014) state that some of the applications used in industries such as airlines, healthcare, gaming, and retail incorporate the graph database since it can serve millions of users.

Consequently, the review study the data migration between different NoSQL solutions, namely, Document-based database (MongoDB), Columnar NoSQL (Azure Table), and Graph-based (Neo4j). Significantly, the research attempts to model the graph database from document and columnar databases to appreciate the aforementioned graph database advantages.

## 2.4 Data Migration Stratgeies

To appreciate the above inference, it may therefore be advantageous to study the different data migration approaches which shall also encourage to deduce the novel approach for the research problem.

Thalheim & Wang (2013) illustrate the phenomenon of data migration which consists of mainly three stages, namely, Extract, Transform, and Load (ETL).

1. Extract: At high-level of abstraction, a legacy data source is first extracted.
2. Transform: At this stage, a legacy data source is transformed into new data structure which may also involve validation, mapping, and cleansing.
3. Load: A new data structure is loaded into target data source.

In context, Scavuzzo, Di Nitto & Ceri (2014) suggest the two general strategies in order to migrate the data:

1. Direct Mapping: Translation of source database into the target database without any intermediate stage.
2. Intermediate Mapping: When data from source data store is first translated into an intermediate (metamodel) format and then transformed into the data structure of target database.

Further, Scavuzzo, Di Nitto & Ceri (2014) state that although an extra transformation occurs in second approach, but it would be much flexible and extensible strategy for data migration. Indeed, the review advocates the use of intermediate mapping approach in order to perform data migration between two different NoSQL solutions.

On the other hand, Thalheim & Wang (2013) enumerate several other approaches for efficient data migration such as

- Big bang: A strategy which takes over all operational data in single execution after transforming the source data into target database.
- Chicken little: In this approach, a source data is divided into small modules and these modules are migrated one-by-one.
- Butterfly: This strategy uses crystalliser to translate the source data into target after freezing the source database.

In contrast, Shirazi, Kuan & Dolatabadi (2012) use the design pattern to perform the data migration between Columnar NoSQL (HBase) and Graph Database (Neo4j). Moreover, Shirazi, Kuan & Dolatabadi (2012) state that design pattern supports in designing an appropriate schema for database. In addition, design pattern also shall address the key concerns in migration and performance such as data integrity, and query efficiency.

As a matter of fact, the review intends to design an extensible migration system which shall exhibit intermediate mapping. Therefore, the research proposes an metamodel which defines the intermediate format and supports the characteristics of NoSQL stores, Columnar, Document, and Graph-based.

# Chapter 3

# Design

## 3.1 Specifications

In an attempt to migrate the data across heterogeneous NoSQL data stores in cloud, this section underlines several strategies to achieve optimal performance. Further, the research presents a novel approach to migrate the data between different NoSQL solutions, specifically, document-based and graph database after comparing several performance-oriented algorithms.

The research models a data migration framework which aims to migrate the data from document database to graph database, and indirect migration using columnar database as intermediate source/target between document and graph-based database. Since these NoSQL solutions advocate the different data models, the research designs an algorithm which includes the data standardization, classification, and staging.

In order to accomplish the data migration process, the research highlights the background for the aforementioned stages involving certain key steps such as extraction, translation, and conversion. Further, the algorithm constitutes the complete NoSQL migration framework using metamodeling which acts as a generic template for easy translations.

Since the research attempts in migrating the document based database to another format, the source database (document database) is first scanned. To appreciate the novel mechanism, the algorithm performs the document parsing to identify the data types involved. Further, depending on the target database, the scanned data is transferred through described metamodel representation. Suppose the target database is graph-based, then the scanned data is stored in processing stores.

Further, the approach explains the translation and mapping of the source database with target database. Post to appropriate mapping, the source data and data structure will be analyzed to retrieve properties, entities, entity key, and column. Further, the direct translator will translate the retrieved data into generic format which could be then used to convert to any requested NoSQL data store, for instance, if a user wants to migrate the data from MongoDB to Azure Table, in case, data from MongoDB will be first converted to metamodel.

Subsequently, an inverse translator will transform the metamodel data into the data structure of requested target data store. Finally, the translated data is saved into target database. But before saving the data into target data store, it is required that metamodel must support formats for every NoSQL data stores. For instance, MongoDB and other document-based databases such as DocumentDB, CouchDB supports JSON, BSON, and XML formats with multi-valued data types whereas Azure Table encourages single valued data types only. Importantly, the algorithm defines the source as well as target database format before actual migration.

After the migration of data from document database to columnar database, the framework designs an algorithm to translate the columnar datastore to graph database. During this translation, the primitive and referential columns of columnar database are scanned. In addition, the hierarchical nodes using referential columns and corresponding relationships are established.

Furthermore, the research defines the direct transformation from document database to graph-based database. The direct transformation includes the direct mapping and data staging which adds the referential attributes of source database. After the addition of referential nodes, the algorithm describes the properties, and relations in graph database.

Consequently, the research studies several aspects concerning data integrity, and consistency. As a proof of concept, the proposed solution establishes the novel mapping between heterogeneous NoSQL solutions which further results in efficient data migration.

## 3.2 Design Overview

As the research presents the approach for the data migration across cloud based different NoSQL data stores, this section presents the system overview which shall highlight the best approach that conforms to our research goal. Importantly, the proposed solution to the research problem also introduces the effective strategy to preserve the data consistency and integrity.

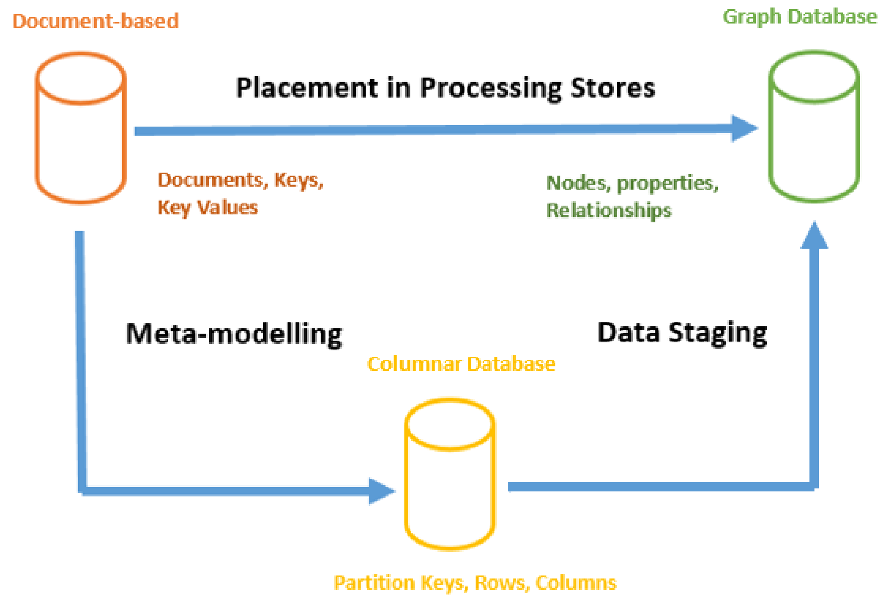



Figure 3.1: System Layout

Consequently, the following study provides the high level detail of data migration framework across heterogeneous data formats. But before learning the working of system, it is important to understand the key elements such as extractor, parser, metamodel, and generator, etc.

Further, Figure 3.1 represents the system architecture which involves a few logical stages: extraction, parsing/translation, meta-modeling, identification and data staging. In brief, in an extraction phase, a extractor retrieves the data from source data store which then is passed to Parser. Further, a Parser translates the extracted data into metamodel format (a standard data format).

Post to meta-modeling, an algorithm extracts the data from intermediate target and identifies the data records/entites. After identification, the algorithm defines an appropriate data format for sequential processing of data records and the data is saved to target data store.

## 3.3 Document Database to Graph-based Database

The research presents the novel approach to migrate the data directly from document-based NoSQL database to graph-based database, for instance , from MongoDB (BSON based database) to Neo4j (graph) database.

But before studying the data migration framework, we must learn the key components which supports the novel translation between two heterogeneous data stores. In context, the algorithm defines the elements such as extractor, document parser, and generator which constitutes complete data migration.

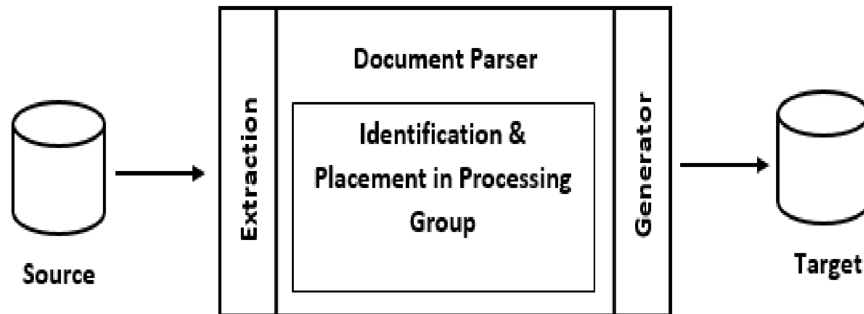



Figure 3.2: Document to Graph-based Database

Figure 3.2 presents the high-level overview of the complete data migration which involves the aforementioned components. To illustrate, the data is extracted sequentially from the source data store in order to ensure the data integrity. Since the research aims at migration of complete database from document style to graph version, the algorithm first identifies the number of collections and retrieves all collections sequentially.

**Algorithm 1:** Collection Classification

```
Pre-Condition:  Database must be instantiated
1.  method classifyCollection(Database)
2.  collection ← classifyUserCollection(Database)
3.  systemCollection ← classifySystemCollection(Database)
4.  return collection
```

Using algorithm 1, collections are classified into the user collections (created or imported by users) and the system collections (created by host database). After classification, the data records (document in case of Document-based database) are processed, the document parser classifies the data according to the data types, for instance, document may contain sub-documents, arrays, and single-valued data types.

Note: A document may describe hierarchical data such as documents within documents, arrays within documents as well as documents within arrays.

**Algorithm 2:** Document Classification

```
Pre-Condition:  Document database must be instantiated
1.  method classifyMainDocument(Collection)
2.  SubDocument ← classifySubDocument(MainDocument)
3.  Array ← classifyArray(MainDocument)
4.  Single ← classifySingle(MainDocument)
5.  return elements
```

Importantly, the document-parser identifies and categorizes the data records (documents), in other words, document parser seggregates the data such as arrays, documents, and single data type values recursively. Post to parsing the data, an algorithm 2 stores the data according to their classification type and passes the records to generator.

The generator retrieves the classified data from parser and generates the graph. Therefore, it may be advantageous to study the generation of graph which exhibits nodes, relationships, and related properties.

### 3.3.1 Data Transition to Graph

To appreciate the novel approach to translate the data from document-based NoSQL database to Graph database, this section explains the high level steps required to produce the graph database.


```
{
    "_id": {
        "$oid": "55c336f040e9cb16aa6931b9"
    },
    "Phone": "+1 (888) 546-3417",
    "friends": [
        {
            "id": "0",
            "name": "Jordan Bradshaw"
        }
    ],
    "Email": "slaterlogan@suretech.com",
    "Company": "SURETECH",
    "Name": {
        "Fname": "Dr. Slater",
        "Lname": "Logan"
    },
    "Gender": "male"
}
```

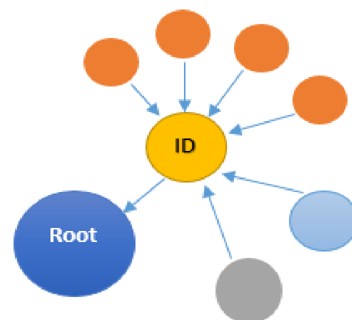




Figure 3.3: Data Modeling in Document and Graph-based Databases

Figure 3.3 represents the two NoSQL databases which model the data in different ways, for instance, the document database stores the data in XML, YAML, JSON or BSON format whereas the graph database uses nodes, relationships, and properties for data

storage. Further, the algorithm ensures the mapping of document elements to graph nodes and properties as follows:

Table 3.1: Document Mapping with Graph

| Document Datastore | Graph Database |
|---|---|
| Database Name | Root Node |
| Collection Name | Sub-root node |
| Document's ID | Entity Node |
| Key | Sub-Node |
| Referred Key | Hierarchical Sub-node |

Post to mapping of major attributes of document database with graph database, the algorithm invokes the generator module. The generator performs recursive (to obtain hierarchical data) parsing, and creates the nodes each containing key name and value, further, these nodes are connected to origin node. And these connections describe the relationships between nodes with appropriate property.

After connecting the collection (sub-root) nodes with root node (database), the fundamental keys are attached to the ID node whereas referencing keys are parsed recursively. Subsequently, the referencing keys are scanned and added to the ID nodes in hierarchical manner.

**Algorithm 3:** Translation to Graph

```
Pre-Condition:  Document must be classified
1.  method generateGraph(Collection)
2.  subRootNode ← generateRoot(getCollectionName)
3.  mapSubRoot(subRootNode,collectionName)
4.  mapEntityNode(idNode,Document.id)
5.  mapSubNodes(subNodes,Document.keys)
6.  mapReferenceNodes(referringNodes,DocumentReferencing.Keys)
7.  return graph
```

In order to perform translation, the algorithm 3 creates the root node using the database name (a database which contains several documents) and connects the collection nodes to database (root node). Subsequently, the ID nodes (using ID of document) to distinguish the different documents are connected to collection node. These IDs are obtained from document in database using document parser as discussed earlier.

Further, the algorithm 3 sequentially parses the document recursively to seggregate the multi-valued types into single valued data types and presents the data in a hierarchical way. Consequently, the data can be migrated efficiently from document-based database to graph database with the help of recursive parsing and placement of data

in hierarchical format to form an easy network model.

The research also presents the novel NoSQL data migration framework which involves two translations: document database to columnar database and columnar database to graph database.

## 3.4 NoSQL Data Migration Framework

To appreciate the data migration phenomenon, the research attempts to describe the translations of data from document style to columnar format and columnar database to graph based database.

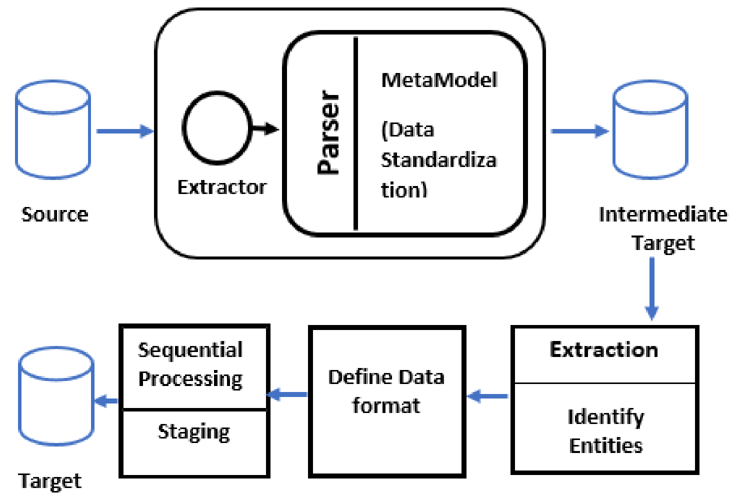


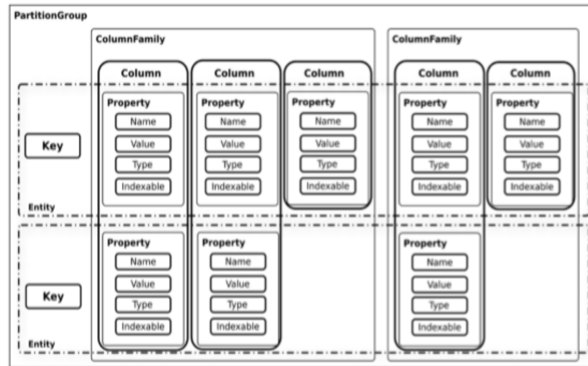

Figure 3.4: NoSQL Framework

### 3.4.1 Document-based to Columnar Database

In this part of the study, the research translates the source (Document-based) to intermediate target (Columnar) database. From figure 3.4, the data migration is performed using intermediate mapping as described earlier.

- Direct Mapping: Translation of source database into target database format,
- Intermediate Mapping: An intermediate model between source and target databases, i.e. source databse is first translated into generic format and then, intermediate formatted data is translated into data structures of target databse.

The research advocates an intermediate mapping which involves aforementioned components for data migration and supports the large number of transformations. To illustrate, a algorithm first extracts the data from source NoSQL data store (Document-based) using extractor which is usually in XML, JSON, YAML, or BSON format. After retrieving the documents, an extractor transfers the data to the parser which significantly categorize the data according to the data types and stores in standardized data format.




A meta-model (a standard data format) stores the data entities alongwith properties such as name, value, and data type. Therefore, an parser extracts the keys with corresponding values and places the entites into meta-model.

Subsequently, these entities are retrieved from meta-model representation and sequentially stored in intermediate target data store. However, Shirazi, Kuan & Dolatabadi (2012) suggest another technique to migrate the data using cloud brokers. Moreover, Shirazi, Kuan & Dolatabadi (2012) attempt to create an abstract layer by integration of multiple cloud provider's resources. Further, this abstract service will be provided by cloud broker to customers, thereby, making all resources transparent. In addition, Shirazi, Kuan & Dolatabadi (2012) specify that design patterns could be used to migrate the data from NoSQL column family to NoSQL graph database. Importantly, there are two main constraints in migration:

1. Just as the foreign key role in RDBMs, there is probability that a specific column of the column family database may refer to another row.

2. Difficult to support the probability to identify appropriate one among several version of a specific column in a row generated using timestamp.

In contrast, the algorithm stores the keys and values with consistently examining the entities and column family. Consequently, Scavuzzo, Di Nitto & Ceri (2014) defines some of the key components of metamodel:

**Property:** Stores the characteristics of datum with its name; also defines the type of datum and boolean attribute (indexable or not).

**Entity and Key:** The properties related to same element are grouped together and referred as Entity. Entity Key is used to distinguish several entities containing properties.

**Column:** Similar properties referring to different entities are grouped together, and known as column.

**Partition Group:** Users are free to construct the data model according to the need, for instance, GAE uses ancestor paths, and Azure uses a combination of table name and partition key.

### 3.4.2 Data Translation

During this stage, two aforementioned translators: direct and inverse translator performs translations in order to migrate the data from one data store to data store. According to Scavuzzo, Di Nitto & Ceri (2014), partition key and table name are combined before mapping to the metamodel partition group.

| | | UserRatings | | |
|---|---|---|---|---|
| **Row Key** | **Partition Key** | **Timestamp** | **email** | **topicName** |
| marco@polimi.it | pk1 | 1392973421 | marco@polimi.it | Politics |
| giovanni@polimi.it | pk2 | 1392985432 | giovanni@polimi.it | Music |
| paola@polimi.it | pk1 | 1392996718 | paola@polimi.it | Politics |

Figure 3.5: Columnar Data Model

After mapping the source database, for instance, columnar in fig.3.5, to the metamodel as discussed earlier, it is required to transform the metamodel representation into target data store format. Thus, an inverse translator transforms the metamodel into the data structure used by target data store.

However, Scavuzzo, Di Nitto & Ceri (2014) raise a few issues while translation such as strong consistency, increased overhead of writes. Though, to ensure consistency, entities of the same partition group in metamodel could be mapped to entities referring to the same ancestor path. A fig.3.6 represents the migrated data from Document to Columnar database using metamodel, which would also allow to migrate the data among other NoSQL based cloud platforms such as Amazon DynamoDB etc.

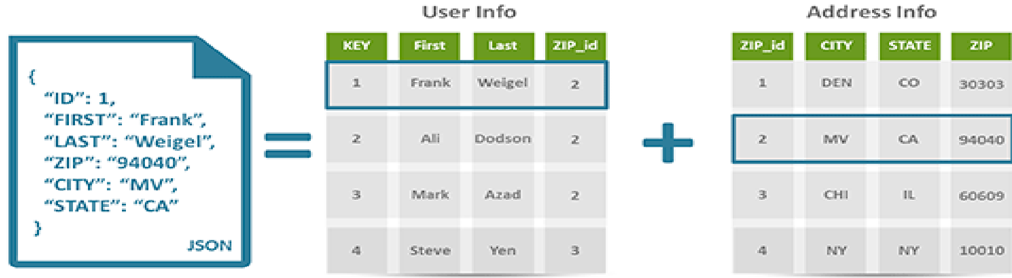



Figure 3.6: Data Translation Overview

**Algorithm 4:** Translation to Columnar Database

```
Condition:  Parsing and Meta-modeling must be completed
1. method mappingToColumnar(entity)
2. partitionKey ← mapPartitionGroup(metamodel, DatabaseName)
3. pKey ← mapPartitionGroup(metamodel, collectionName)
4. rowKey ← mapEntityKey(metamodel, documentID)
5. columnName ← mapPropertyKey(metamodel, keys)
6. rowValues ← mapPropertyValue(metamodel, KeyValue)
7. return metamodel
```

An algorithm 4 describes the translation of NoSQL document database to NoSQL columnar database after executing the document parsing and metamodel formation.

Hence, the proposed solution for the efficient data migration would exhibit extensibility among columnar NoSQL data stores and also ensures data consistency and integrity.

### 3.4.3 Columnar NoSQL to Graph-based NoSQL

After the migration of data from document-based NoSQL store to columnar NoSQL, the research attempts to migrate the data from columnar NoSQL to graph-based NoSQL.

It may therefore be advantageous to study the graph-based database in order to ensure the adaptability of data to target from columnar source. According to McKnight (2014*c*), graph-based databases are data stores which exhibits hierarchical or network model to store the data and allows navigation. Significantly, McKnight (2014*c*) enumerates several components of graph database:

**Node:** A node is an entity which defines a pair such as name and value, moreover, there can be multiple nodes (and of multiple types) in a single graph.

**Properties:** All nodes or relationship among them can describe specific properties (attributes), relationships are connections between different nodes.

**Path:** A path defines the address where an application persists the data as well as the routes between nodes which are used for traversal.

Before translating the database, the algorithm establishes the mapping between columnar database and graph database as follows:

Table 3.2: Columnar and Graph Database Mapping

| Columnar Datastore | Graph Database |
|---|---|
| Partition Name | Root Node |
| Referential partition | Sub-root Node |
| Row ID | ID-Node |
| Primitive Columns | Sub-node |
| Reference Columns | Hierarchical-node |

Subsequently, the algorithm 4 extracts the data records sequentially from columnar database and identifies the entities in order to define the appropriate data format.

For instance, columnar data store uses partition key and row keys, thus, the algorithm defines the root node (a main node to which all other nodes are connected) with the partition key property and connects the row keys (ID entities) to the defined root node recursively.

**Algorithm 5:** Transforming to Graph Database

```
Condition:  Columnar object must be instantiated.
1.  method fromColumnarStore(Entity)
2.  rootEntity <- generateRoot(entity.partitionKey)
3.  mapPartitionGroup(rootNode,rootEntity)
4.  mapRootNode(rootNode,rootEntity.Value)
5.  mapRowEntity(rowNode,entity.rowkey)
6.  mapColumns(subNodes,entity)
7.  return mainGraph
```

In order to ensure consistency, the algorithm 3.5 performs sequential processing of data records:

- Entities from columnar database are described as nodes with values,
- Further, the algorithm describes the relationship to represent the connection between entities, for example, an entity Name can be classified as First-Name and Last-Name,
- Finally, the node and relationship properties which are entity attributes (metadata such as timestamp) are described.

While performing the sequential processing, the algorithm also introduces the data staging which identifies an additional entities stored in columnar database.

During the data staging, the reference columns of database are examined and entities are filtered to form the hierarchical nodes. Finally, the research models a dynamic filter which filters the primitive and reference columns. After identifying the reference columns, the partition groups corresponding to the columns are scanned.

Subsequently, the referencing entities are retrieved and mapped to hierarchical nodes in graph database. Indeed, the research implements the designed algorithm in chapter 4 which shall further demonstrate the different NoSQL solutions and involved translations.

# Chapter 4

# Implementation

This chapter discusses the implementation and development strategies of the cloud based NoSQL migration framework in detail which supports data migration across heterogeneous NoSQL data models such as document, columnar, and graph based database. The NoSQL data migration framework has been developed using the Application programming Interfaces (APIs) provided by MongoDB, Microsoft Azure, and Neo4j in java archive format. Therefore, it would be beneficial to study the various java APIs for different databases used in implementing the migration framework.

## 4.1 Data Standardization

In order to implement the data migration, it is essential to standardize the source data and classify the data according to the data type. The research demonstrates the migration of data from MongoDB (as document-based source database) to Azure Table (columnar- intermediate target database) and Azure Table to Neo4j (graph-based target database), importantly, a direct data translation from MongoDB to Neo4j i.e. document to graph database. According to Kanade, Gopal & Kanade (2014), MongoDB is an open source, and document based schema free database which models the data in BSON (Binary encoded JSON data), also, it exhibits the features such as auto-sharding, map-reduce, hybrid deployments in cloud.

To appreciate the migration framework, the algorithm models two main strategies:(i) BSON Parsing and (ii) Meta-Modeling in order to standardize and classify the data from MongoDB. In this section, the research entails the different APIs and various methods defined to implement the aforementioned strategies for efficient data migration.

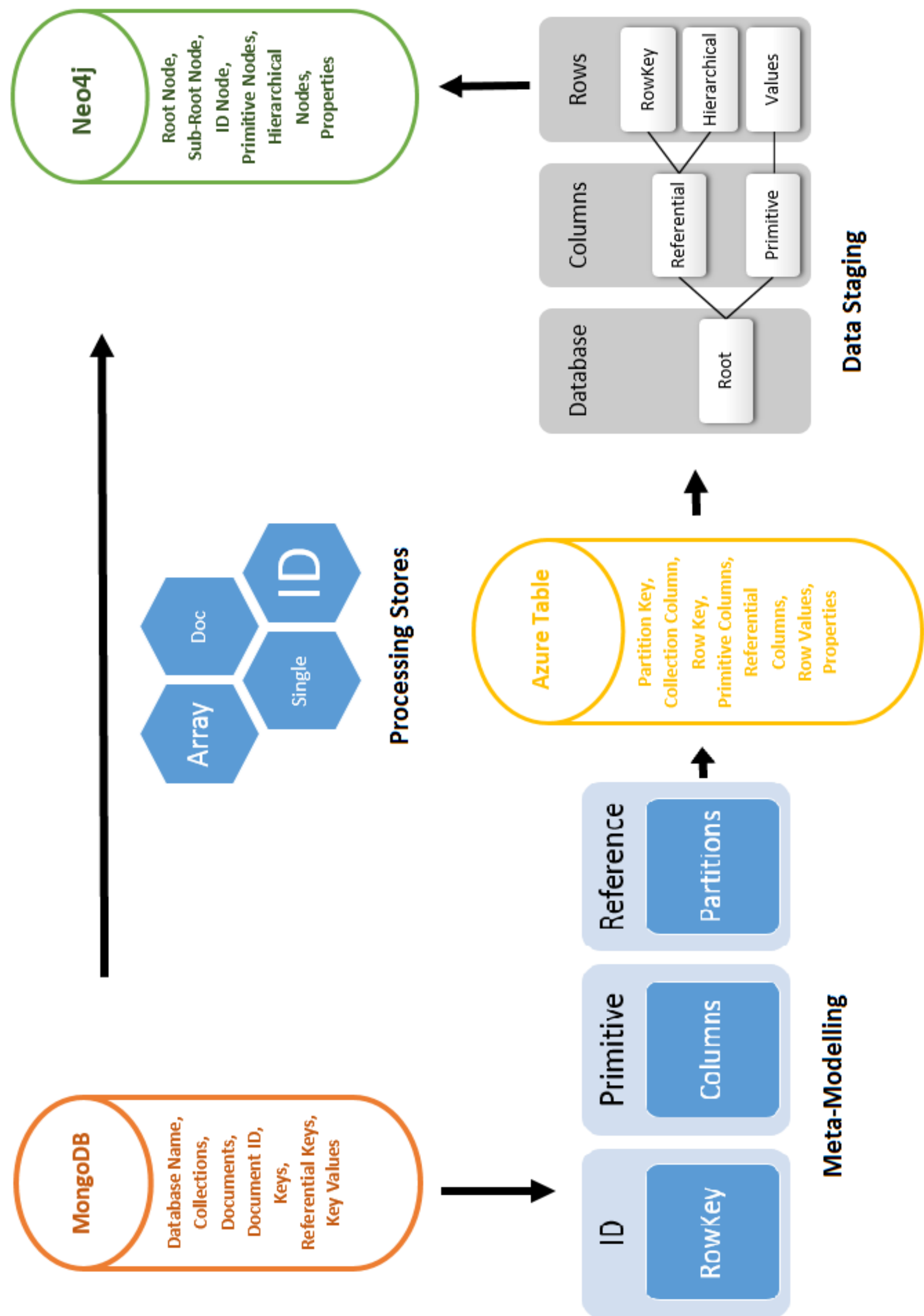



Figure 4.1: System Implementation

But before modeling the migration strategies, it is important to establish an appropriate authenticated and authorized connection with source data store. Therefore, the research attempts to deploy the MongoDB instance on Microsoft Azure which can support large volume of data. In context, the MongoLab `https://mongolab.com/` provides a fully managed MongoDB service that can be easily deployed on Microsoft Azure (a public cloud), in addition, mongolab `http://docs.mongolab.com/`allows the application to access the data using shell commands or APIs/drivers as can be seen in fig.4.2.

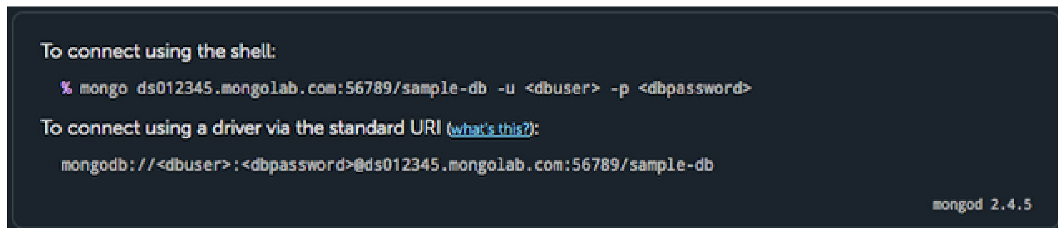



Figure 4.2: MongoLab Connection

After establishing the authenticated connection with MongoLab hosted on Azure, it is required to import the data (which is supposed to be migrated), therefore, we imported a few large BSON data files (from appendix A.1) . Note: prior to data import, we must create database and collection (which contains several documents). In order to model the real-world system, the research advocates the import of real data set, thus, real json data from `http://data.gov.uk/data/search` is imported. Subsequently, the algorithm uses MongoDB libraries such as `mongodb-async-driver-2.0.1.jar,` `mongo-java-driver-3.0.2.jar`, and `mongodb-binding-0.2.0.jar` in order to access the data from MongoDB.

The algorithm involves the several java classes such as `BasicDBObject,` `DBCollection,` `DBCursor,` `MongoClient,` and `MongoClientURI` which are provided by aforementioned libraries. These pre-defined java classes ease the development and retrieval of data from MongoDB.

As discussed in chapter 3, the research attempt to migrate the complete database from one NoSQL format to another. Therefore, the algorithm first seggregates the different collections constituting the document-based MongoDB database. MongoDB contains two types of collections: User and System, the algorithm performs all processing on user-based collections.

Since the research designs an approach which processes the data records sequentially, thus, the algorithm performs parsing of data records and stores in metamodel respectively in order to migrate the data from document database to any other format.

### 4.1.1 Document Parsing

As MongoDB supports the data in a BSON format (BSON is superset of JSON), thus, the algorithm models a novel BSON parser which recursively seggregates the data.

Prior to learning the BSON parser, it is essential to study the different classes and libraries provided by `bson-2.9.0-sources.jar` such as `BsonArray, BsonDocument, BsonType, and BsonValue`, etc. which support BSON parsing. These classes supports in distinguishing the BSON data according to the data type such as Array, Document and primitive. Subsequently, the BSON based parser scans the document (data record) and classifies the data elements of document recursively.

```
/* method called recursively to identify the sub-documents inside document */
private HashMap<String,HashMap> parseDoc(String key, String string, DBObject ob)
  {
    HashMap<String, String> mapTest = new HashMap<String, String>();
    HashMap<String, HashMap> mapMain = new HashMap<String, HashMap>();
    BsonDocument b1 = BsonDocument.parse(ob.toString());
    Set<String> a1 = b1.keySet(); // obtaining keys
    for(String as:a1)
    {
      System.out.println(" values: "+ob.get(as).toString()+" of key "+as);
      String val = ob.get(as).toString();
      mapTest.put(as, val);
    }
    mapMain.put(key, mapTest);
    // returning the parsed document (keys and values)
    return mapMain;
  }
```

Listing 4.1: Parsing Sub-documents in document

The program code 4.1 presents the method which performs parsing of sub-documents which further contains several keys and values. In addition, the method stores the keys and values after mapping to the super-document key in metamodel. Further, the obtained keys and values are scanned to determine whether they are single-valued (primitive) or multi-valued data elements.

If the obtained data is multi-valued, the data is scanned recursively else it is mapped to the super-document key and stored in meta-model or processing store depending on target database as shown in Figure 4.1. In context, the algorithm determines the different possible data types as shown in program code 4.2:

```
1 // to identify the Sub-document inside document
2 (s11.equalsIgnoreCase(s.getKey())) && ((String.valueOf(s.getValue().getBsonType(). ↩
      toString()).equals("DOCUMENT")))
3 // to identify the Array within document
4 (s11.equalsIgnoreCase(s.getKey())) && ((String.valueOf(s.getValue().getBsonType(). ↩
      toString()).equals("ARRAY")))
5 // to determine the single valued data types
6 s11.equalsIgnoreCase(s.getKey())
```

Listing 4.2: Parsing Conditions

The algorithm attempts to seggregate the documents until the data is refined to single-valued elements. Alongwith seggregation, the single-value data is stored and mapped to super data type key. For instance, if the parser encounters an array inside document, then, the array is parsed further to obtain the inside data elements. Further, these array elements are scanned to determine the data type of each element, thus, the parser continues scanning recursively depending on the data type. Therefore, the data elements are refined to single-value elements with their corresponding keys and stored in meta-model.

### 4.1.2 Meta-modeling

Post to parsing and segregating the data elements, it is essential to maintain the relationships between super and sub keys/values as well as the data-types. Consequently, the algorithm imports the library `java.util.*` which provides collection and other related classes used for meta-modeling. Since, the part of the NoSQL migration framework attempts to migrate the data from document to column-based (Azure Table) database, the algorithm implements the meta-model for appropriate mapping with target.

Subsequently, the metamodel implements the following key points:

1. MongoDB database contains different collections is mapped to partition key in meta-model.

2. Further, the collections (attributes) are mapped to collection columns in Azure Table using metamodel.

3. Document's 'id' key is directly mapped to meta model entity key which is later stored as Row key in Azure Table.

4. Other keys and values are mapped to property names (column names) and property values (row values) respectively.

Hence, the meta-model (using Figure 4.1) eases the migration of data from document to columnar database as well as maintains the data integrity since the data propogated is serialized first. After migration of the serialized data to the destination database, data is deserialized using java utilities and methods.

Further, it is important to understand the translation algorithm which transforms the meta-model to Azure Table considering the generic algorithm described in chapter 3.

**Algorithm 6:** Mapping to Azure Table

```
Condition:  Parsing and Meta-modeling must be completed
1. method mappingToAzureTable(entity)
2. partitionKey ← mapPartitionGroup(metamodel, MongoDB-DatabaseName)
3. CollectionColumn ← mapCollection(metamodel, MongoDB-Collection)
4. rowKey ← mapIDKey(metamodel, documentID)
5. columnName ← mapEntityKey(metamodel, keys)
6. rowValues ← mapEntityValue(metamodel, KeyValue)
7. return metamodel
```

To illustrate, the algorithm 6 explains the method which is implemented in order to translate the documents (data records) to columns entities sequentially, but it is required that meta model must be instantiated prior to migration.

**Algorithm 7:** Mapping MongoDB Values to Azure Table Rows

```
Condition:  Keys are translated to Columns
1.  method fromMetamodel(metamodelValues)
2.  if existsReferentialEntity(metamodelValue.ReferentialBSON) then
3.  newPartitionKey ← createPartitionKey(metamodelValue.BSONType))
4.  for each column in metamodelValue do
5.  rowValue ← new metamodel(referenceColumn,metamodelKey.Value)
6.  mapColumns(azureRowValue, metamodelValue)
7.  return rows
```

Importantly, the algorithm 7 describes the iteration of meta-model which possesses the keys and values for the document retrieved from MongoDB database. The algorithm creates the new partition if any new partition entity i.e. referential document from the same collection is found. Similarly, if new key is traversed in metamodel, then a new column is introduced in Azure Table to ensure the absolute migration with consistency.

Further, the figure 4.3 describes the implementated class and methods for performing the aforementioned algorithms. Also, the methods represented in UML class diagrams are executed to form the meta-model.

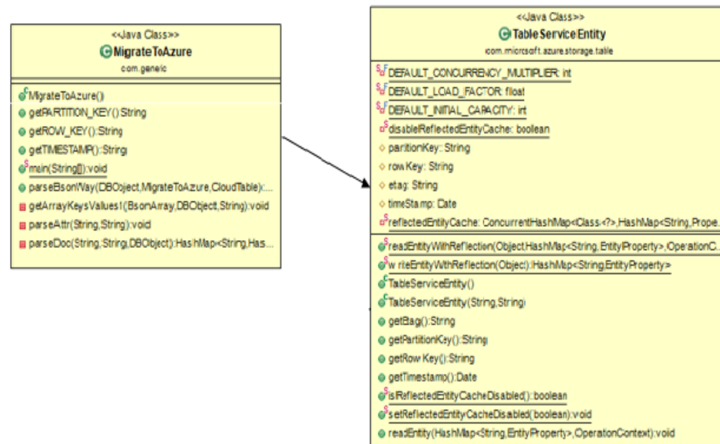



Figure 4.3: UML Diagram for Azure Migration

An important aspect which is taken into account is absolute consistency i.e. the framework ensures that the each partition key contains the entities in same table.

Note: While traversing the data elements or keys from the source database, one read and two writes operations are performed which may effect translation time but ensures data consistency.

## 4.2 Data Classification and Staging

After migrating the data from document (MongoDB) to columnar (Azure Table) database (see appendix A.3), the framework attempts to transform the data from columnar database to graph-based data store. Consequently, the research implements Neo4j as a NoSQL graph database which allows the framework to create tens of billions of nodes and relationships as discussed in chapter 3.

Significantly, in order to implement the data migration from columnar to graph database, it is important to study the data classification and staging. The data stored in Azure Table must be traversed sequentially and placed in an appropriate processing group. In other words, the algorithm categorizes the row values obtained from Azure Table to create the graph which enables easy traversing and easy to understand.

Subsequently, the algorithm first establish the authenticated connection with Azure Table using Microsoft Azure Storage APIs. For instance, `azure-core-0.7.0.jar, azure-management-network-0.7.0.jar, and azure-management-storage-0.7.0.jar, etc.` provides multiple classes to access

the Azure Table in java environment.

```
1  /*.................. Azure Table Storage ..........................*/
2  CloudStorageAccount storageAccount =
3                 CloudStorageAccount.parse("DefaultEndpointsProtocol=http;" +
4                    "AccountName=abc;" +
5                  "AccountKey=QxeDp0ZFW5d8XTWB9/ ↩
                      a7hbT6cbmwES7WmxxTtLPfPisDwGgbEve1JyeiJxzY//u1ZXtCyEDyxA==");
6    CloudTableClient tableClient = storageAccount.createCloudTableClient();
7
8    CloudTable smallSample = tableClient.getTableReference("sampleData");
9    smallSample.createIfNotExists();
10 /*................ End of Azure Table Storage Creation...............*/
```

Listing 4.3: Azure Connection

After establishing the connection with Azure using programe code 4.3, the algorithm scans the partition groups formed using partition keys. These partition keys seggregate the entities in Azure Table according to the different collections (in MongoDB). After obtaining the partition key, the framework introduces a dynamic filter which allows to iterate the entities related to particular row key. In brief, a java class `DynamicTableEntity` enables the collection filter and row filter to retrieve the related column values for that row key. Also, the algorithm performs the data staging which creates the nodes correspondingly, therefore, it would be advantageous to study the graph implementation.

### 4.2.1 Neo4j - Graph Implementation

In this section, the research discusses a part of the algorithm which deals with graph implementation. But before learning the implementation, it is important to study the sevaral APIs provided by Neo4j which can be used in Java environment. For instance: `neo4j-graph-algo-2.2.3.jar, neo4j-graph-matching-2.2.3.jar, neo4j-import-tool-2.2.3.jar, and neo4j-io-2.2.3.jar`, etc. provides several useful classes such as `GraphDatabaseService, Label, Node, Relationship, RelationshipType, Transaction, and GraphDatabaseFactory`, etc.

Subsequently, the algorithm utilizes the aforementioned libraries and classes to implement the graph database. In terms of implementation, GraphDatabaseService class can be used to specify the database path (directory where graph database is stored) as well as it provides several methods to support Transaction class to start and shutdown

the transactions. Further, the Node, and Relationship classes can be used to create the nodes and define the relationships among the associated nodes respectively.

Importantly, the research hosts the Neo4j on Microsoft Azure (virtual machine) and deployed a cloud service to store graph database.

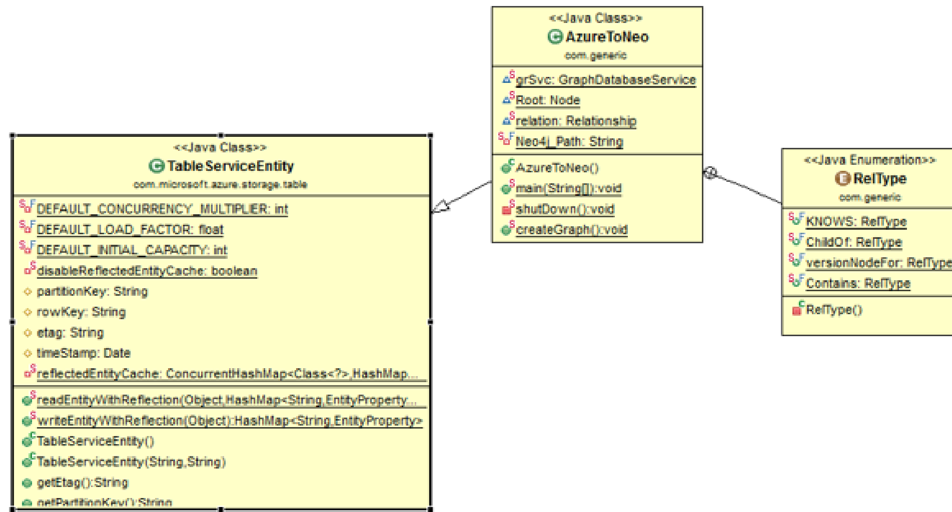



Figure 4.4: UML Diagram for generation of Graph Database

The fig.4.4 represents the dependencies among the classes and enum prototype which further describes the different type of nodes and relationships used to create the graph. The main class `AzureToNeo` retrieves the data records from MongoDB hosted on MongoLab (at Microsoft Azure) by extending `TableServiceEntity`. The class `TableServiceEntity` enables the sub class to retrieve `partition key, row key, and entities` from Azure Table, moreover, `DynamicTableEntity` supports `TableServiceEntity` to retrieve the records dynamically. After retrieving the entities from Azure, the algorithm sets the data path using `GraphDatabaseService` as discussed earlier. Prior creating the graph, it is important to initialize the main nodes, sub-nodes, and relationships using Neo4j class methods and enumeration. Consequently, the algorithm initializes the nodes, relationships, and Relationship Types such as `KNOWS, Contains, ChildOf, etc.`

The algorithm 8 explains the generation of root node, and aforementioned sub nodes, thus, it would be beneficial to study the translation of row values into sub nodes after creating root node. For instance, partition key from Azure Table is mapped to root node and collectionColumn to sub-root node as well as row key is mapped to 'id' node in graph. Subsequently, the primitive columns (`Name, Title, Notes, Type`) are mapped to sub nodes but it is also important to translate the referred columns such as Reference1 and Reference2. Therefore, the proposed graph algorithm traverses the

**Algorithm 8:** Transforming Azure Table to Neo4j Graph

```
Condition:  Azure Table object must be instantiated.
1.  method fromAzureTable(Entity)
2.  rootNode ← generateRoot(entity.partitionKey)
3.  mapPartitionGroup(sub-rootNode,entity.collectionColumn)
4.  mapSubRootNode(sub-rootNode,collectionColumn.Value)
5.  mapRowEntity(IDNode,entity.rowkey)
6.  mapColumns(subNodes,entity.primitiveColumns)
7.  return mainGraph
```

referred columns in Azure to establish hierarchical structure in Neo4j.

Consequently, the program code 4.4 illustrates the working of dynamic filter and building of graph from referential columns. The referential columns are scanned using row keys in different partitions since multiple partitions can exhibit same row keys.

```
1   String partitionFilter1 = TableQuery.generateFilterCondition(PARTITION_KEY, ↩
        QueryComparisons.EQUAL,pk);
2   TableQuery query1 = new TableQuery().take(1);
3   TableQuery<MigrateToAzure> partitionQuery1 = TableQuery.from(MigrateToAzure.class) ↩
        .where(partitionFilter1);
4
5   for (MigrateToAzure entity1 : cloudTable.execute(partitionQuery1))
6     {
7         String rowFilter1 = TableQuery.generateFilterCondition(ROW_KEY, ↩
              QueryComparisons.EQUAL,entity1.getRowKey());
8           /* .........combining two filters..........*/
9           Iterable<DynamicTableEntity> rslt12 = cloudTable.execute(query1.from( ↩
                DynamicTableEntity.class).where(partitionFilter1));
10
11              if((entity1.getRowKey()).equalsIgnoreCase(entity.getRowKey()))
12                  {
13                    for(DynamicTableEntity d1:rslt12)
14                    {
```

Listing 4.4: Dynamic Entity Filter

Significantly, the graph algorithm defines the relationships alongwith the instantiation of nodes with the appropriate properties. Consequently, the algorithm 9 describes the phenomenon of defining relationships among main nodes and sub-nodes. Moreover, the algorithm defines the properties that enable the users to view the value of the selected node or relationship at runtime.

After instantiating the primitive and referential nodes and relationships in graph

**Algorithm 9:** Nodes and Relationships

```
Pre-Condition:  GraphDataService must be enabled.
1.  Create Root Node
2.  map(idNode, rowKey)
3.  method createMainRelationship(RootNode, idNode)
3.1.  relation ← idNode.createRelationship(RootNode, RelationshipType)
3.2.  relation.setProperty("relationship", "Unique entity")
4.  create sub-Node
5.  method createSubRelationships(idNode, sub-Node)
5.1.  relation ← sub-Node.createRelationship(idNode, RelationshipType)
5.2.  relation.setProperty("relationship", "sub-nodes")
```

database, the proposed framework adds the properties to the nodes. During traversal, the algorithm performs the data staging i.e. adding the additional or hierarchical nodes depending on the processing node in order to ensure data consistency.

```
1  else if(set.equalsIgnoreCase("friends"))
2  {
3    Node reference = grSvc.createNode();
4    Node id, f_name;
5      int no = entity.getTheme-primary();
6    reference.setProperty("Friends", no);
7    for (int k=1;k<=no;k++)
8    {
9      Node ent = grSvc.createNode();
10     relation = ent.createRelationshipTo(reference, RelType.ChildOf);
11     relation.setProperty("relationship", "resources");
12     ent.setProperty("Extras", k);
13     id = grSvc.createNode();
14     theme = grSvc.createNode();
15     if(k==1)
16     {
17       id.setProperty("ID", entity.getId1());
18       theme.setProperty("", entity.getTheme-primary1());
19       }
```

Listing 4.5: Data Staging

In context, the program code 4.5 shows the data staging while traversing the referenced columns in Azure Table. To illustrate, when an algorithm encounters the referred columns in Azure Table, it iterates the entities until the column size. During this iteration, several sub nodes are created (as many entities) and added to the parent node (main reference node). Correspondingly, the relationships and properties are established at runtime to describe the hierarchical graph database which ensures

consistency in data structuring.

Note: Retrieving data records or entities from Azure Table and generation of graph nodes/relationships are performed simultaneously to reduce the number of reads, thus, the algorithm may ensure efficiency in migration.

## 4.3 Direct Data Transformer

In this section, the algorithm transforms the document NoSQL database to graph based database i.e. from MongoDB to Neo4j as discussed in chapter 3.

In order to implement the direct transformation, the algorithm parses the BSON based documents from the MongoDB recursively. Therefore, the framework performs the previously described `BSON parsing` in section 4.1.1. Importantly, the algorithm creates different processing stores for each type of data elements present in document as represented in figure 4.1. Consequently, it is important to identify the data elements to ensure the appropriate placement of elements in processing store.

The processing stores define the elements type, for instance, data element can be an array type, document type or single-valued type. Suppose if data element is an array type, then again the array may contain `document, array or single-value`. Therefore, the `array` is placed in specific processing store which is scanned during nodes generation, similarly, `sub-document` is stored in different processing group for further execution.

```
1 HashMap<String,HashMap> h1 = buildDocStore(s.getKey(), ob.get(s11).toString(),od, ↩
      docKey); // for Document Type
2
3 HashMap<String,String> buildPrimitive = new HashMap<String,String>();
4 buildPrimitive.put(s.getKey(),ob.get(s11).toString()); // for Primitive Type
5
6 HashMap<String,HashMap> h12 = buildArray(str, dps.get(str).toString(),od); // for ↩
      Array Store
```

Listing 4.6: Building Processing Stores

Considering the programe code 4.6, the algorithm implements the processing stores using `java.util.collection` classes. The methods such as `buildDocStore(), buildPrimitive() and buildArray()` creates the processing stores and retrieves the corresponding keys and values from `BsonParser`.

As a consequence, the algorithms ensures the consistency by processing different types of data elements and enabling appropriate scanning of each sub-element.

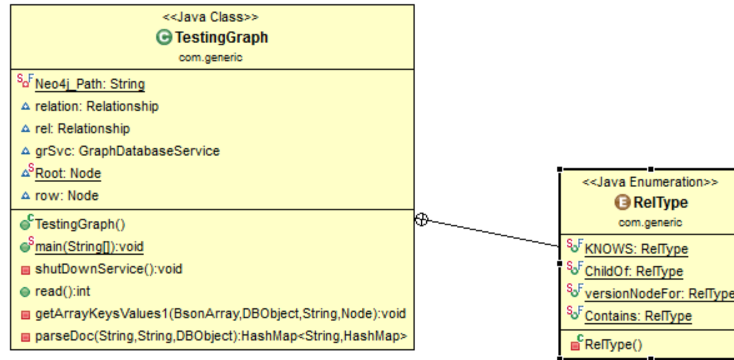



Figure 4.5: UML Diagram for Direct Graph Generation

Figure 4.5 presents a UML class diagram for defined algorithm which produces the graph database from document-based NoSQL database (MongoDB). The figure also shows the different methods for multiple processing stores, for instance, `getArrayKeyValues(), and parseDoc()`. Since the framework is focussing on direct translation, it is important that documents from MongoDB must be scanned sequentially and processed simultaenously. Further, this will enhance the efficiency of data migration which involves direct mapping. Therefore, the following section will explain the implementation of direct mapping between two different databases.

### 4.3.1 Direct Mapping

To appreciate the direct transformation of data, the algorithm retrieves the data elements according to the priorities i.e. first it retrieves the database, collections, document ID's and then all descendents. The approach ensures the proper placement of data elements to the processing group and appropriate mapping.

Table 4.1: Direct Mapping

| MongoDB Datastore | Neo4j Graph Database |
|---|---|
| Database Name | Root Node |
| Collection Name | Sub-root Node |
| Document's ID | ID Node |
| Primitive Keys | Primitive Sub-Nodes |
| Referred Keys | Hierarchical Nodes |
| Key Values | Properties |

The table 4.1 explains the direct mapping between document based MongoDB and graph based Neo4j database. In terms of implementation, database name from MongoDB is mapped to root node of Neo4j graph. Further, the collections in MongoDB are

seggregated based on system and user defined criteria. Each collection contains certain set of douments with unique ID, thus, the IDNodes are generated and connected with collectionNode. After learning the mapping of elements, the algorithm performs certain set of operations to save the data in graph format, as following:

**Algorithm 10:** Generating Graph

```
Pre-Condition:  MongoDB must be instantiated
1.  method generateGraph(Collection)
2.  sub-rootNode ← generateRoot(getCollectionName)
3.  mapSubRoot(sub-rootNode,collectionName)
4.  mapSubRootNode(idNode,DocumentID)
5.  mapSubNodes(subNodes,Document.keys)
6.  mapReferenceNodes(hierarchicalNodes,Document.ReferentialKeys)
7.  return graph
```

The algorithm 10 maps as well as saves the data in form of nodes and establishes the relationships same as defined in algorithm 9. Moreover, the steps in algorithm `4, 5, and 6` are repeated recursively in order to build the hierarchical structure for referencing nodes (see appendix A.4). In addition, the properties alongwith the nodes are stored directly which are retrieved during scanning the keys in document.

```
1 Label docLabel = DynamicLabel.label(s11);
2 docKey.addLabel(docLabel);
3 docKey.setProperty("DocValue", s11);
4 relation = docKey.createRelationshipTo(entities, RelType.ChildOf);
5 relation.setProperty("relationship",s11+" is a child of Entity");
```

Listing 4.7: Establishing relations and properties

The program code 4.7 describes the implementation of establishing relations and properties of the nodes. As a proof of concept, the framework involves efficient mapping and standardization concepts which ensures the data consistency in migration. Consequently, the NoSQL database migration framework enables the user to migrate the data across heterogeneous NoSQL solutions.

# Chapter 5

# Evaluation

In this chapter, the research performs a few tests/experiments to verify the behavior and performance of the designed NoSQL data migration framework. Since the framework involves three data translations between heterogeneous NoSQL solutions (document-based, columnar, and graph database), the research reports several relevant evaluations.

Consequently, the research uses the real json dataset (see appendix A.2) available on UK government site (`http://data.gov.uk/dataset/england-national-crime-mapping`) to evaluate the migration system. The dataset describes the crime and neighbourhood policing information which can be accessed using police API for crime statistics.

Further, the research focusses on the evaluation of the (1) overhead (system performance), (2) total migration time, and the (3) throughput of framework in data transitioning from:

(i) Document-based (MongoDB) to Columnar (Azure Table)

(ii) Columnar (Azure Table) to Graph database (Neo4j)

(iii) Document-based (MongoDB) to Graph database (Neo4j).

Also, the research monitors the CPU usage in execution of aforementioned data transitions. Hence, the overall time duration is calculated i.e. the time required to complete the entire migration. Importantly, the research studies the variance of performance of executed data transitions.

Before learning the evaluation of NoSQL data migration framework, it is important to examine the compatibility between heterogeneous NoSQL solutions.

## 5.1 Compatibility Tests

Since the chapter 4 discusses the implementation of NoSQL data migration framework using MongoDB, Azure Table, and Neo4j as document store, columnar, and graph database respectively, this section highlights a few compatibility aspects which must be considered before an experiment.

The first evaluation aims to address the different modeling prototype (format, size, and attributes) supported by MongoDB, Azure Table, and Neo4j.

### 5.1.1 MongoDB vs Azure Table vs Neo4j

In order to deduce the differences, the research refers the documentation provided by these organizations. Subsequently, `http://docs.mongodb.org/master/reference/limits/` illustrates the MongoDB limits and thresholds, for instance, MongoDB supports the 16 MB as maximum size of any BSON document and 100 levels of nesting per document only. Further, the MongoDB models eventual consistency under which the system needs not to reflect the latest writes but changes propagate gradually. MongoDB uses namespace exhibiting certain attributes (namespace length, number of namespaces, and size of namespace file), for instance, 16 MB namespace file may support nearly 24000 namespaces which includes index and collection.

Likewise, the Neo4j documentation available at `http://neo4j.com/developer/guide-performance-tuning/` explains the Neo4j data modeling which stores the graph data (e.g. nodes, properties, and relationships)in a number of different store files such as `neostore.nodestore.db (N), neostore.propertystore.db (P), and neostore.relationshipstore.db (R)`. Moreover, the site entails the memory configuration guidelines which are necessary to counter the heap space overhead.

| Neo4j Properties | Metrics |
|---|---|
| neostore.nodestore.db.mapped_memory | (14 * N) / (1024 * 1024)M |
| neostore.relationshipstore.db.mapped_memory | (34 * R) / (1024 * 1024)M |
| neostore.propertystore.db.mapped_memory | (41 * P) / (1024 * 1024)M |
| neostore.propertystore.db.strings.mapped_memory | (128 * P) / (1024 * 1024)M |

Table 5.1: Neo4j Memory Mapping Calculations

Using Table 5.1, the research determines the heap space to allocate the Neo4j graph

processing:

```
heap (H) = Total memory - OS memory (OS) - total mapped memory (MM)
```
(5.1)

Further, the site https://msdn.microsoft.com/en-us/library/azure/dd179338.aspx provides the documentation explaining Azure Table attributes. The site highlights that the Azure Table supports an entity with only 255 properties, which includes 3 system properties, thus, the user can define 252 properties. Also, the data size of all entity's properties can not exceed 1MB. Furthermore, the partition keys and Row keys used mustn't exceed 1KB size because Azure Table supports the string value for these keys upto 1KB only.

Consequently, the different NoSQL models present the compatibility issues and challenges in data migration. So, in a particular case, the NoSQL data migration framework should be modeified to appreciate the original data types and properties.

## 5.2 Cloud scenario

As the NoSQL data migration system is executed in cloud environment, the tests are performed on the Microsoft Azure platform. An azure platform supports many programming languages such as .NET, Java, PHP, etc. as well as selection of different OS, tools, databases, frameworks. Azure provides VHD (Virtual Hard Disk) which can be pre-defined, or user-defined in order to create virtual machine (VM). An user can specify the number of cores and amount of memory with each VM.

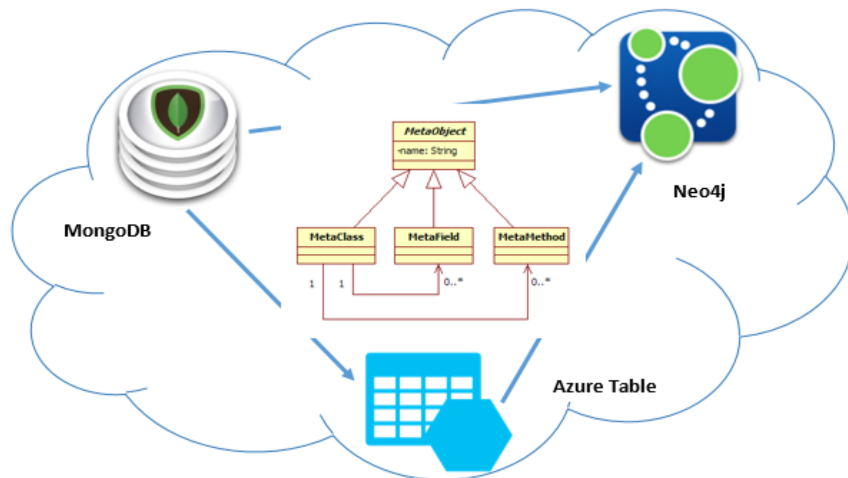



Figure 5.1: Deployment Architecture of NoSQL Migration Framework

Subsequently, to conduct the tests, using synthetic benchmarking: a virtual machine (VM) with eight virtual cores and 14GB of RAM is created. Further, the virtual machine is physically hosted by Azure in Western-Europe. The virtual machine runs a Windows Server 2012 Datacenter, and the test environment has been configured:

(i) log4j library is integrated in the NoSQL migration system which provides the details such as overall time requirement for migration completion, and throughput of each translation,

(ii) Azure monitor provides a certain relevant information concerning migration, for instance, disk read and write (Bytes/second), CPU percentage, etc.

Using Figure 5.1, all three databases, namely, `MongoDB, Azure Table, and Neo4j` were hosted on virtual machine created on Azure platform. Further, the translation models (metamodel, parsing, and processing stores) are deployed as cloud service.

## 5.3 Document to Columnar Database

Since the research implements the direct data transformation from document database to columnar database using section 4.1, this part of the study evaluates the data migration process from MongoDB to Azure Table.

To appreciate the working of NoSQL data migration framework, the research conducts the five tests concerning the migration of data from MongoDB to Azure Table, each with different number of data records (documents) to be migrated. Based on MongoDB and Azure Table compatibility, an average size of MongoDB document is 1024 Bytes. Hence, the appropriate json data from http://data.gov.uk/dataset/england-national-crime-mapping has been considered to translate 16MB, 32MB, 64MB, 128MB, 256MB, and 512 MB data.

| Metrics | set#1 | set#2 | set#3 | set#4 | set#5 |
|---|---|---|---|---|---|
| Source size(MB) | 16 | 64 | 128 | 256 | 512 |
| Records | 16384 | 65536 | 131072 | 262144 | 524288 |
| Migration Time(sec) | 1281 | 1946 | 3551 | 6433 | 12051 |
| Throughput(records/s) | 12.7 | 33.6 | 36.9 | 40.7 | 43.5 |
| Avg. CPU Usage (%) | 4.2 | 5.3 | 8.7 | 10.1 | 14.5 |

Table 5.2: MongoDB to Azure Table

Using table 5.2, it can be seen that with the increment in source size, the migration time accelerates linearly. Importantly, the throughput of the migration system i.e. data

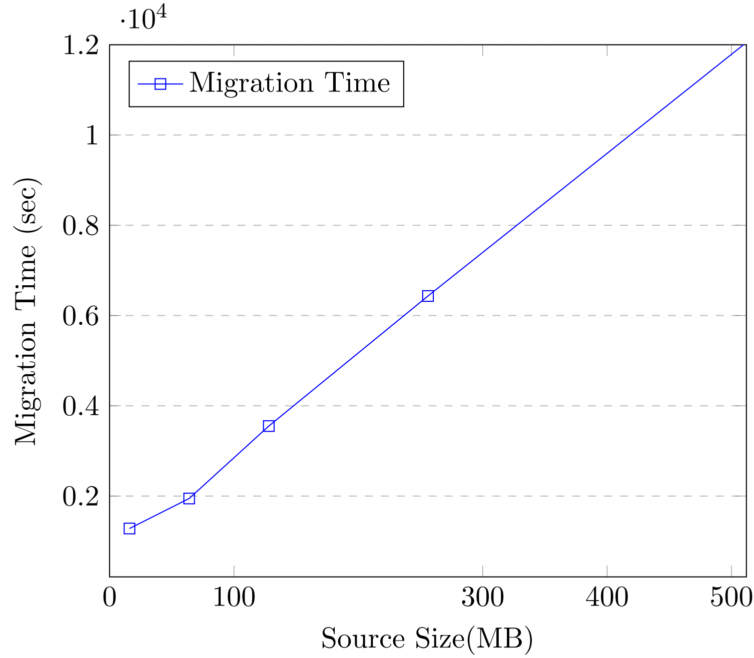



Figure 5.2: Migration Time vs Source Size

records per second increases with the increasing data records (documents). Further, Figure 5.2 presents the increment in migration time which involves extraction, translation, and writing time to target datastore with respect to different source sizes. Also, it is noted that the migration time increases significantly while migrating 16MB, and 64MB data. Subsequently, the average CPU usage in migrating the source data also shows the similar pattern. Note: The average CPU usage (%) lies between 4% and 15% depending on source dataset size while performing the migration from MongoDB to Azure Table.

## 5.4 Columnar to Graph Database

In this section, the research attempts to examine the same metrics, namely, `Migration Time, Throughput, and average CPU usage`. Taking section 4.2 into account, an aforementioned metrics will be evaluated while migrating the five different dataset sizes (`16MB, 64MB, 128MB, 256MB, 512MB`) from Azure Table to Neo4j.

Considering Table 5.3, it can be seen that migration time increases drastically with the increase in source dataset size. In comparison to previous migration (MongoDB to Azure Table), the column to graph database shows poor performance. **Since the migration from Azure Table to Neo4j involves filtering and staging of entities**

| Metrics | set#1 | set#2 | set#3 | set#4 | set#5 |
|---|---|---|---|---|---|
| Source size(MB) | 16 | 64 | 128 | 256 | 512 |
| Entities | 16384 | 65536 | 131072 | 262144 | 524288 |
| Migration Time(sec) | 1553 | 2961 | 5938 | 10395 | 17183 |
| Throughput(records/s) | 10.54 | 22.13 | 22.07 | 25.21 | 30.51 |
| Avg. CPU Usage (%) | 5.3 | 7.9 | 11.8 | 17.5 | 24.8 |

Table 5.3: Azure Table to Neo4j

**while MongoDB to Azure Table exhibits only metamodeling**. Due to this reason, the throughput of the migration system does not enhance radically. Further, there was significant gain in average CPU consumption in migrating the different size datasets (especially large datasets) from Azure Table to Neo4j comparatively.

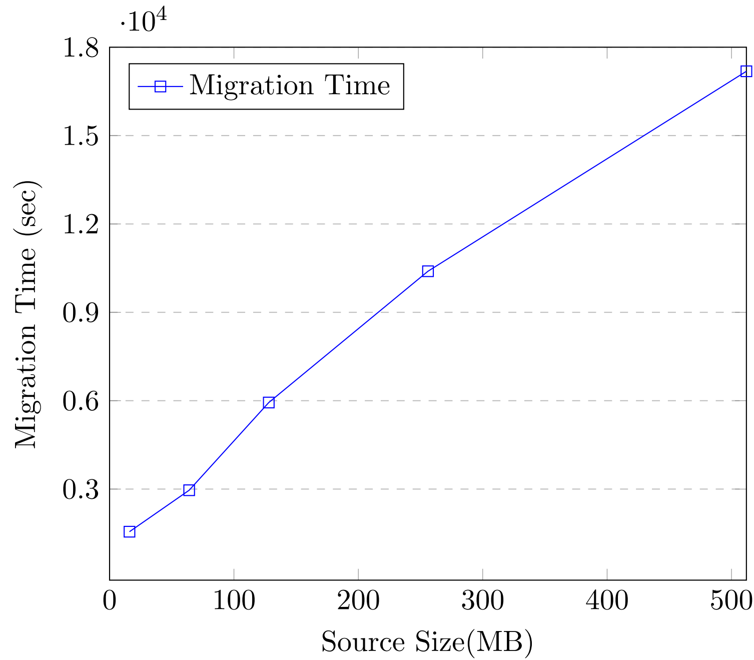



Figure 5.3: Azure Table to Neo4j

Figure 5.3 demonstrates how the migration time on varying the source database size. For instance, the migration time for dataset#2 and dataset#3 i.e. 64MB and 128MB respectively, becomes approximately twice since the throughput of the system remains invariable. In particular, the CPU usage almost got doubled but the outcome (throughput) does not enhance conventionally.

Whilst the above study evaluates the NoSQL data migration framework involving (i) MongoDB to Azure Table, and (ii) Azure Table to Neo4j, it is also important to study the performance of direct transformation of Document database to Graph database.

## 5.5 Document to Graph Database

In order to evaluate the performance metrics, this part of the study takes the section 4.3 into account which implements the direct translation from MongoDB to Neo4j. Further, the section also compares the outcome with afore performed translations.

| Metrics | set#1 | set#2 | set#3 | set#4 | set#5 |
|---|---|---|---|---|---|
| Source size(MB) | 16 | 64 | 128 | 256 | 512 |
| Documents | 16384 | 65536 | 131072 | 262144 | 524288 |
| Migration Time(sec) | 1096 | 2673 | 4796 | 7743 | 11869 |
| Throughput(records/s) | 14.9 | 24.51 | 27.32 | 33.85 | 44.17 |
| Avg. CPU Usage (%) | 3.9 | 5.6 | 8.4 | 14.7 | 19.3 |

Table 5.4: MongoDB to Neo4j

Using Table 5.4, it can be noted that the throughput of the migration system while translating the documents directly into the nodes increases linearly. Subsequently, it takes less time to migrate different datasets from MongoDB to Neo4 in comparison to Azure Table to Neo4j translation. Though the average CPU usage also increases linearly with the throughput yet the system performs better in comparison to data translation from Azure Table to Neo4j.

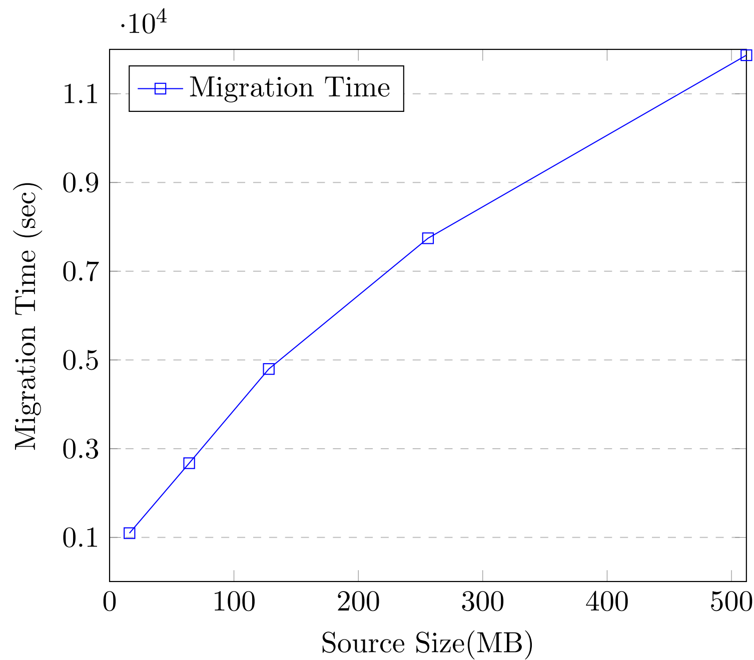



Figure 5.4: MongoDB to Neo4j

Furthermore, on comparing the graphs 5.2 and 5.4, we can deduce that the translation from MongoDB to Azure Table performs better than MongoDB to Neo4j data translation.

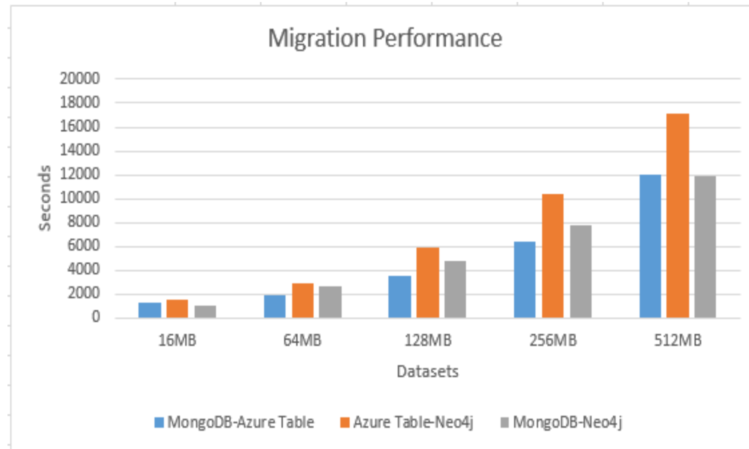



Figure 5.5: Performance of Translations

Figure 5.5 clearly demonstrates the difference in performances of aforementioned translations. To illustrate, for smaller data sets such as `16MB and 64MB`, all the three translations performs nearly same. But when the translations are executed on larger datasets (`128MB, 256MB, and 512MB`), an experiment shows significant performance differences. For instance, while migrating the datasets #4 and #5, a translation from Azure Table to Neo4j was much inefficient in comparison to other two translations.

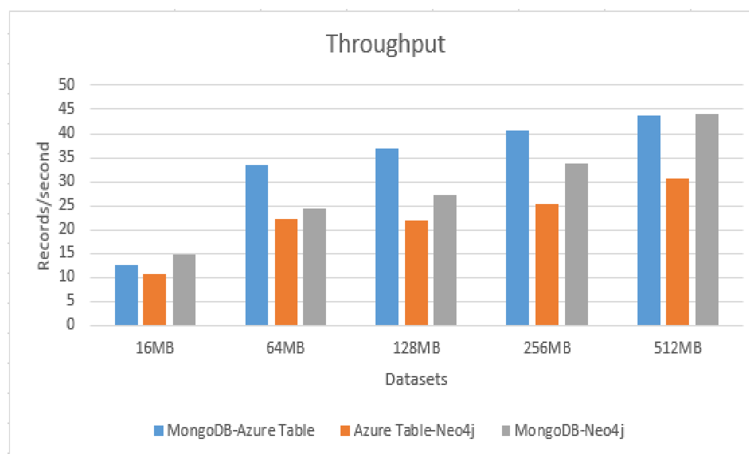



Figure 5.6: Throughput of Translations

To appreciate the comparison between three translations, it is essential to study the throughput variance. Subsequently, figure 5.6 presents the contrast between the throughput of the three different migrations.

As a proof of performance, throughput of the translation from Azure Table to Neo4j is significantly low than other two translations. Also, the CPU overhead was maximum in translating the data from Azure Table to Neo4j depending on the datasets size. Consequently, the NoSQL migration framework demonstrates the best performance while migrating the database from MongoDB to Columnar, however, the direct translation from MongoDB to Neo4j performs more efficiently for large datasets.

The research also evaluates the performance of query execution in order to determine the behavior of all three NoSQL solutions. In addition, it would be beneficial in

## 5.6 Query Performance

In this section, the research studies the variation of query performance in MongoDB, Azure Table, and Neo4j under different dataset sizes (`16MB, 256MB, and 512MB`). In order to evaluate the query performance, the research executes the 'search query' on aforementioned sized datasets. Subsequently, a few unit test classes have been implemented to conduct the Read/Search query tests on all NoSQL solutions.

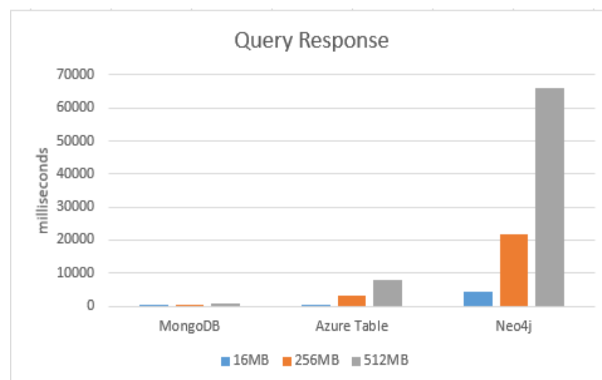



Figure 5.7: Query Performance

A graph in figure 5.7 depicts that when a search query is executed on different sizes of MongoDB database, the results are retrieved much faster than in case of Azure Table and Neo4j. Further, it can be seen Neo4j is inefficient in processing a search/read query in comparison to Azure Table.

Further, it would be interesting to evaluate the performance metrics on NoSQL migration framework in-house which shall highlight the differences between in-house and cloud computation of NoSQL databases.

## 5.7 In-house Scenario

This section discusses the tests which have been performed on an Intel Core i5-4200U Processor @1.6GHz (3M Cache, up to 2.30 GHz) with 6GB of RAM running Windows 8.1. In particular, the NoSQL migration framework executes inside Apache Tomcat application server.

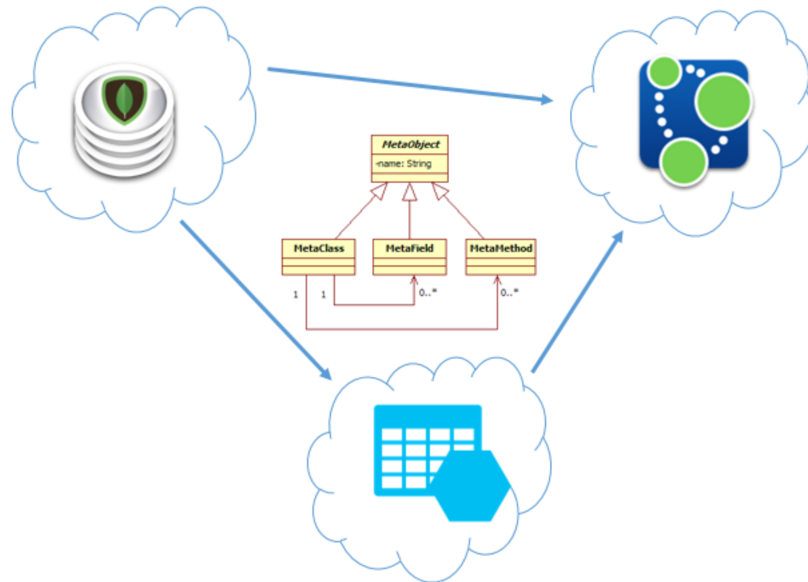



Figure 5.8: In-house Deployment Architecture

The NoSQL migration framework has been deployed on Apache Tomcat while the three databases are hosted on Microsoft Azure platform. Note: Neo4j has been hosted on GrapheneDB (GrapheneDB is a cloud platform for the Neo4j) which further enables the Neo4j to deploy on Azure.

### 5.7.1 In-house Performance

On conducting the same tests, the in-house configuration successfully performs all three translations but fails in migrating the large datasets especially the migrations headed to Neo4j.

Figure 5.9 demonstrates the performance of all three translations, (i) MongoDB to Azure Table, (ii) Azure Table to Neo4j, and (iii) MongoDB to Neo4j. Since the migration of larger datasets (`128MB, 256MB, and 512MB`) failed due to HTTP 500 and heap space overhead, the research evaluates the migration performance on smaller datasets (`16MB and 32MB`). However, the system was able to translate the data (`128MB and`

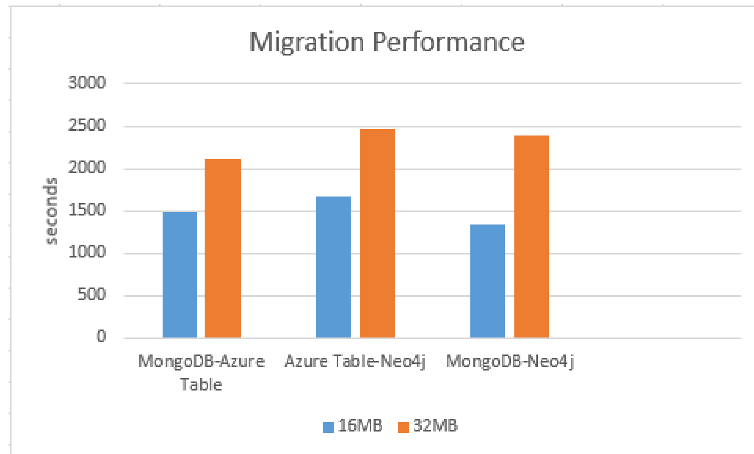



Figure 5.9: In-house Migration Performance

256MB) from MongoDB to Azure Table. Additionally, it has been noted that migration from MongoDB or Azure Table to Neo4j results in maximum CPU overhead as well as memory leakage problem. Subsequently, a memory management method was required to be implemented in order to maximize the migration performance.

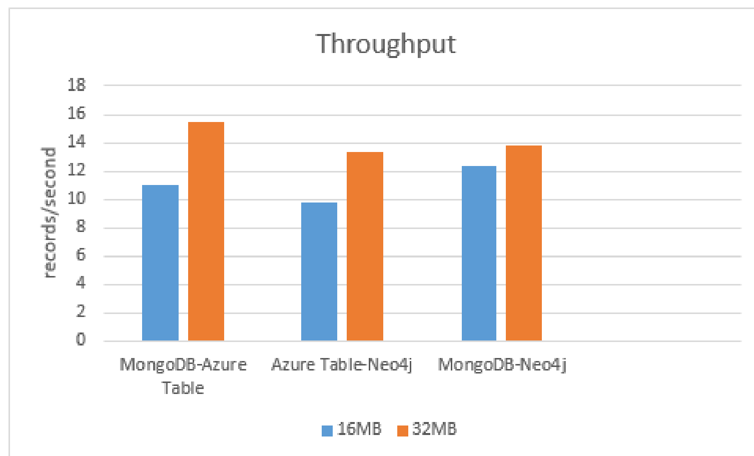



Figure 5.10: In-house Migration Throughput

As can be seen in figure 5.10, the throughput of the NoSQL migration framework also reduced significantly in comparison to cloud based NoSQL data migration. Consequently, the in-house configuration of NoSQL data migration framework is inefficient and not feasible for executing larger datasets.

# Chapter 6

# Conclusion

The research presents a novel approach that enables the migration of data across cloud based heterogeneous NoSQL solutions, in particular, focusing on document, columnar, and graph-based databases. Since there is a broad set of NoSQL implementations as well as each of the NoSQL implementation exhibits different properties and characteristics, the research attempts to address the horizontal heterogeneity among NoSQL databases as explained in chapter 2. Importantly, the chapter 2 discusses the challenges that are required to be addressed while migrating the data on cloud from one NoSQL format to another such as assurance of data adaptability, consistency, and integrity.

In an attempt to eliminate the aforementioned challenges and limitations, the chapter 3 presents the detailed analysis of the state of the art on NoSQL solutions and effective strategies. Furthermore, the research models a NoSQL data migration framework, composed of data standardization and classification strategies. The cloud based NoSQL data migration framework has been designed after considering the latest model driven software engineering techniques which enables an efficient data classification and translation.

To appreciate the data adaptability and consistency, the research implements an efficient strategy to establish the effective mapping between source and target cloud based NoSQL datastore such as Azure Table, MongoDB and Neo4j. Subsequently, the first part of the chapter 4 implements an effective data classification approach which parses the source database and generates the metamodel according to the target NoSQL database. Further, the proposed algorithm maps the metamodel with target database to ensure the data adapatability. Significantly, the algorithm involves a dynamic filtering and data staging strategies which also contributes in preserving data consistency, however, these strategies also influence the performance to an extent.

Whilst it can be seen that the cloud based NoSQL data migration framework addresses the major challenges, the research also attempts to examine and analyse the performance of migration system under different scenarios. To determine the behavior of system, the research has conducted a few relevant tests which determines the efficiency of three translations, namely, MongoDB to Azure Table, Azure Table to Neo4j, and a novel direct transformation from MongoDB to Neo4j. To illustrate, the chapter 5 demonstrates all three translations using different sizes of NoSQL databases such as `16MB, 64MB, 256MB, and 512MB`. Further, the chapter 5 helps in understanding the impact of sequential processing and filtering on performance (migration time) as well as compares the three translation to draw the most efficient NoSQL solution. In other words, it has been evaluated that the document-based database supports efficient migration of data wheras graph database ensures the effective management of huge volumes of data. An experiment also suggests that columnar database could model a more viable strategy to incorporate the referential entities which shall enhance the Read/Write efficiency.

## 6.1 Future Work

Given the pivotal role that data portability plays in eliminating the heterogeneity among cloud based different NoSQL solutions, the research also investigates the three main directives which shall contribute in enhancing the migration system.

**Migration System Extensibility:** The NoSQL data migration framework shall support large number of databases with the addition of new mappings. Further, the extensible system will enable a user to migrate the data on multiple NoSQL solution without any considerable cost and efforts.

**Paradigm shift:** To appreciate the performance of migration, it would be beneficial to introduce map-reduce model which shall support efficient processing and management of large volumes of data. Subsequently, it shall be possible to reduce the huge processing workload and enhance latencies.

**Versioning and lock procedure:** A mechanism shall be integrated with the system which may enable the migration of latest inserted or updated data while ensuring data consistency.

Consequently, the integration of proposed approach with the aforementioned directives may prove beneficial for commercial use and further academic research without any re-engineering process.

# Bibliography

Abramova, V. & Bernardino, J. (2013), Nosql databases: Mongodb vs cassandra, *in* 'Proceedings of the International C* Conference on Computer Science and Software Engineering', C3S2E '13, ACM, New York, NY, USA, pp. 14–22.
**URL:** *http://doi.acm.org/10.1145/2494444.2494447*

Atzeni, P., Bellomarini, L., Bugiotti, F. & Gianforme, G. (2009), 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, ACM, New York, NY, USA, pp. 275–286.
**URL:** *http://doi.acm.org/10.1145/1516360.1516393*

Chandra, D. G. (2015), '{BASE} analysis of nosql database', *Future Generation Computer Systems* **52**, 13 – 21. Special Section: Cloud Computing: Security, Privacy and Practice.
**URL:** *http://www.sciencedirect.com/science/article/pii/S0167739X15001788*

Chen, Z., Yang, S., Zhao, H. & Yin, H. (2012), An objective function for dividing class family in nosql database, *in* 'Computer Science Service System (CSSS), 2012 International Conference on', pp. 2091–2094.

Dharmasiri, H. & Goonetillake, M. (2013), A federated approach on heterogeneous nosql data stores, *in* 'Advances in ICT for Emerging Regions (ICTer), 2013 International Conference on', pp. 234–239.

Frank, L., Pedersen, R. U., Frank, C. H. & Larsson, N. J. (2014), The cap theorem versus databases with relaxed acid properties, *in* 'Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication', ICUIMC '14, ACM, New York, NY, USA, pp. 78:1–78:7.
**URL:** *http://doi.acm.org/10.1145/2557977.2557981*

Gudivada, V., Rao, D. & Raghavan, V. (2014), Nosql systems for big data management, *in* 'Services (SERVICES), 2014 IEEE World Congress on', pp. 190–197.

Kanade, A., Gopal, A. & Kanade, S. (2014), A study of normalization and embedding in mongodb, *in* 'Advance Computing Conference (IACC), 2014 IEEE International', pp. 416–421.

McKnight, W. (2014*a*), Chapter ten - operational big data: Key-value, document, and column stores: Hash tables reborn, *in* W. McKnight, ed., 'Information Management', Morgan Kaufmann, Boston, pp. 97 – 109.
**URL:** *http://www.sciencedirect.com/science/article/pii/B9780124080560000102*

McKnight, W. (2014*b*), Chapter twelve - graph databases: When relationships are the data, *in* W. McKnight, ed., 'Information Management', Morgan Kaufmann, Boston, pp. 120 – 131.
**URL:** *http://www.sciencedirect.com/science/article/pii/B9780124080560000126*

McKnight, W. (2014*c*), Chapter twelve - graph databases: When relationships are the data, *in* W. McKnight, ed., 'Information Management', Morgan Kaufmann, Boston, pp. 120 – 131.
**URL:** *http://www.sciencedirect.com/science/article/pii/B9780124080560000126*

North, K. (2010), 'The nosql alternative', *InformationWeek* (1268), 33–35,38–39. Copyright - Copyright United Business Media LLC May 24, 2010; Last updated - 2015-03-09; CODEN - INFWE4; SubjectsTermNotLitGenreText - United States–US.
**URL:** *https://ezproxy.ncirl.ie/login?url=http://search.proquest.com/docview/347837209?accountid=103381*

Qi, M. (2014), Digital forensics and nosql databases, *in* 'Fuzzy Systems and Knowledge Discovery (FSKD), 2014 11th International Conference on', pp. 734–739.

Ranabahu, A. & Sheth, A. (2010), Semantics centric solutions for application and data portability in cloud computing, *in* 'Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on', pp. 234–241.

Sattar, A., Lorenzen, T. & Nallamaddi, K. (2013), 'Incorporating nosql into a database course', *ACM Inroads* **4**(2), 50–53.
**URL:** *http://doi.acm.org/10.1145/2465085.2465100*

Scavuzzo, M., Di Nitto, E. & Ceri, S. (2014), Interoperable data migration between nosql columnar databases, *in* 'Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW), 2014 IEEE 18th International', pp. 154–162.

Sellami, R., Bhiri, S. & Defude, B. (2014), Odbapi: A unified rest api for relational and nosql data stores, *in* 'Big Data (BigData Congress), 2014 IEEE International Congress on', pp. 653–660.

Shirazi, M., Kuan, H. C. & Dolatabadi, H. (2012), Design patterns to enable data portability between clouds' databases, *in* 'Computational Science and Its Applications (ICCSA), 2012 12th International Conference on', pp. 117–120.

Tauro, C., Ganesan, N., Easo, A. & Mathew, S. (2013), Convergent replicated data structures that tolerate eventual consistency in nosql databases, *in* 'Advances in Computing and Communications (ICACC), 2013 Third International Conference on', pp. 70–75.

Thalheim, B. & Wang, Q. (2013), 'Data migration: A theoretical perspective', *Data and Knowledge Engineering* **87**(0), 260 – 278.
**URL:** *http://www.sciencedirect.com/science/article/pii/S0169023X12001048*

# Appendix A

# NoSQL Data Migration Framework

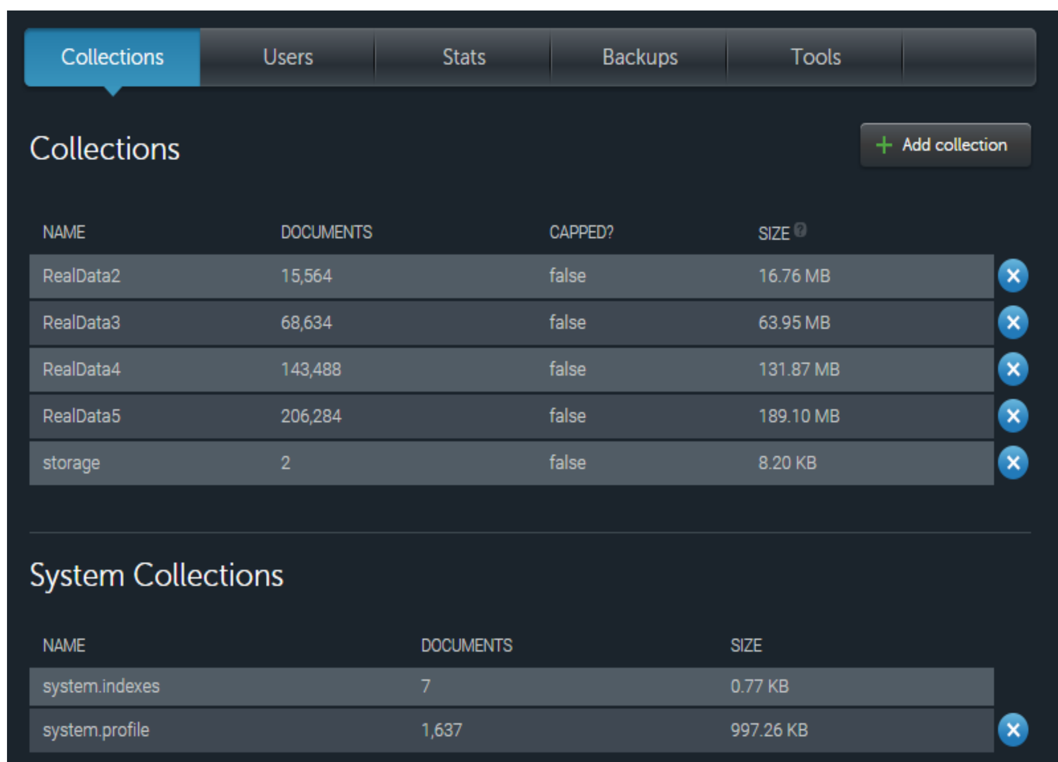



Figure A.1: MongoLab Database on Windows Azure

```
{
    "_id": {
        "$oid": "55ce098680ae4f840f6f35e6"
    },
    "resources": [
        {
            "resource_group_id": "bc6935a0-ec2b-4a5d-9891-23108e5aafd7",
            "url": "http://www.bhamsouthcentralccg.nhs.uk/2012-08-22-13-24-55/expenditure",
            "id": "8f034e7c-f137-4536-ba05-84e7a7975af2"
        }
    ],
    "state": "active",
    "type": "dataset",
    "private": "false",
    "title": "04X >25K",
    "name": "04x-25k",
    "tags": [
        "Health",
        "Environment"
    ],
    "extras": {
        "theme-primary": "Government Spending"
    },
    "ckan_url": "http://data.gov.uk/dataset/04x-25k",
    "notes": "04X Monthly Report >25K"
}
```


Figure A.2: Real dataset (MongoDB Document)

| PartitionKey | RowKey | CollectionName | Name | Notes | Private1 |
|---|---|---|---|---|---|
| MongoLab-9 | 55ce06b280ae2aafe207775b | RealData2 | 05p-25k-july-13 | 05P > 25K July 2013 | false |
| MongoLab-9 | 55ce06b280ae2aafe207775d | RealData2 | 05p-december-25k-expenditure | Solihull CCG >25K Dec Expenditure | false |
| MongoLab-9 | 55ce06b280ae2aafe207775f | RealData2 | 060-data-stats | Ethnicity data | false |
| MongoLab-9 | 55ce06b280ae2aafe2077762 | RealData2 | 1-100-000-offices-points-shapefile | Evidence - Scientific & Evidence Services. Enviro | false |
| MongoLab-9 | 55ce06b280ae2aafe2077763 | RealData2 | 1-250000-soils-and-nsis-wms | This service is the digital (vector) version of the S | false |
| MongoLab-9 | 55ce06b280ae2aafe2077765 | RealData2 | 1-50-000-scale-gazetteer | 1:50 000 Scale Gazetteer provides an excellent re | false |
| MongoLab-9 | 55ce06b280ae2aafe2077767 | RealData2 | 1-class-settlement-morphology | Rural and Urban Definitions Grid showing settler | false |
| MongoLab-9 | 55ce06b280ae2aafe2077769 | RealData2 | 1-in-100yr-maximum-flood-depth | Flood and Coastal Risk Management - Incident I | false |
| MongoLab-9 | 55ce06b280ae2aafe207776b | RealData2 | 10-class-settlement-morphology | Rural and Urban Definitions Grid showing settler | false |
| MongoLab-9 | 55ce06b280ae2aafe207776d | RealData2 | 10cm--50cm-near-infrared-ni-digital-aerial-photography | Flood and Coastal Risk Management - Incident I | false |
| MongoLab-9 | 55ce06b280ae2aafe207776f | RealData2 | 10m-generalised-digital-elevation-model | Evidence - Scientific & Evidence Services.<br>The 10m Generalised Digital Elevation Model (D | false |
| MongoLab-9 | 55ce06b280ae2aafe2077771 | RealData2 | 15th-percentile-house-prices-land-registry | House market data for local authority districts ba | false |
| MongoLab-9 | 55ce06b280ae2aafe2077773 | RealData2 | 16-to-19-bursary-fund-annual-mi-return | The 16-19 Bursary Funding annual data return is | false |

Figure A.3: Real data on Azure Table

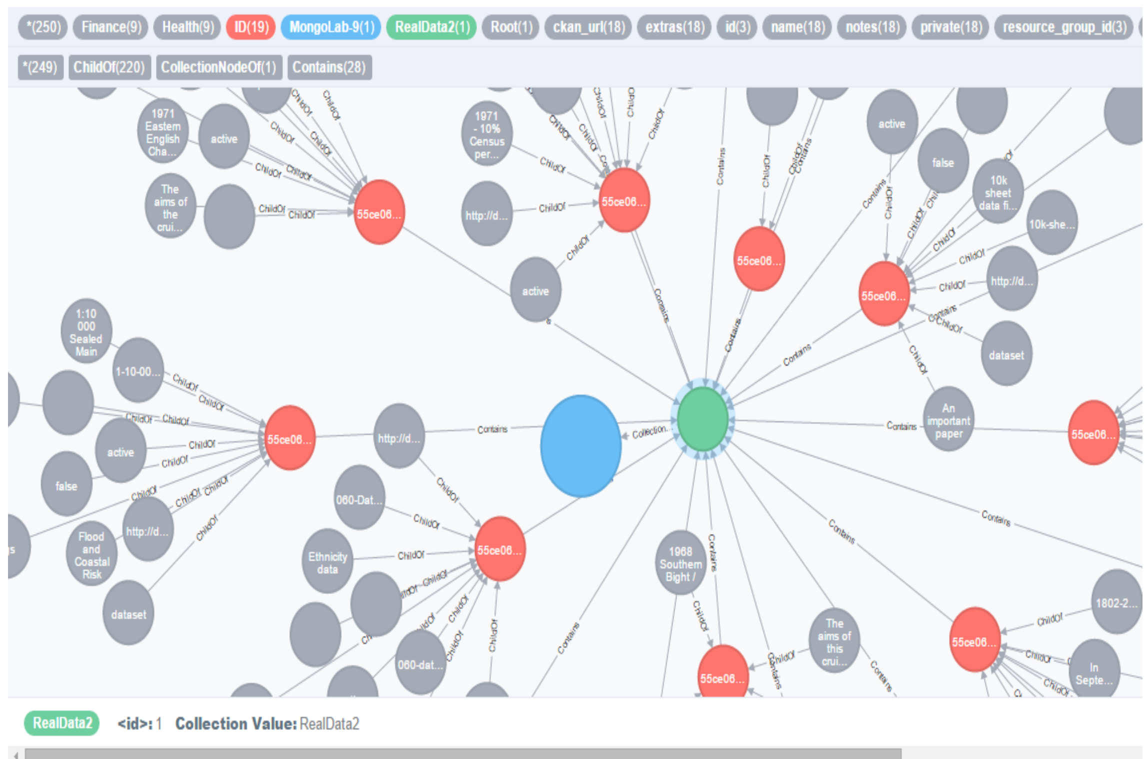



Figure A.4: Real data on Neo4j