

# Transactions and Data Management in NoSQL Cloud Databases

---

**Adewole Conrad Ogunyadeka**

**November, 2016**

Department of Computing and Communications Technologies

Faculty of Technology, Design and Environment

Oxford Brookes University

A dissertation submitted to the Faculty of Technology, Design and Environment in  
partial fulfilment of the requirements of the award of Doctor of Philosophy.

# Abstract

NoSQL databases have become the preferred option for storing and processing data in cloud computing as they are capable of providing high data availability, scalability and efficiency. But in order to achieve these attributes, NoSQL databases make certain trade-offs. First, NoSQL databases cannot guarantee strong consistency of data. They only guarantee a weaker consistency which is based on eventual consistency model. Second, NoSQL databases adopt a simple data model which makes it easy for data to be scaled across multiple nodes. Third, NoSQL databases do not support table joins and referential integrity which by implication, means they cannot implement complex queries. The combination of these factors implies that NoSQL databases cannot support transactions. Motivated by these crucial issues this thesis investigates into the transactions and data management in NoSQL databases.

It presents a novel approach that implements transactional support for NoSQL databases in order to ensure stronger data consistency and provide appropriate level of performance. The novelty lies in the design of a Multi-Key transaction model that guarantees the standard properties of transactions in order to ensure stronger consistency and integrity of data. The model is implemented in a novel loosely-coupled architecture that separates the implementation of transactional logic from the underlying data thus ensuring transparency and abstraction in cloud and NoSQL databases. The proposed approach is validated through the development of a prototype system using real MongoDB system. An extended version of the standard Yahoo! Cloud Services Benchmark (YCSB) has been used in order to test and evaluate the proposed approach. Various experiments have been conducted and sets of results have been generated. The results show that the proposed approach meets the research objectives. It maintains stronger consistency of cloud data as well as appropriate level of reliability and performance.

# Declaration

Some parts of the work presented in this thesis have previously appeared in the following published paper:

Ogunyadeka A, Younas M, Zhu H, Aldea A. "A Multi-Key Transactions Model for NoSQL Cloud Database Systems", Proc. of the 2nd IEEE International Conference on Big Data Computing Service and Applications (BigDataService 2016), Oxford, England, UK, 29 March -1 April 2016.

# Acknowledgements

I would like to express my deepest gratitude and appreciation to my supervisor and Director of Studies Dr. Muhammad Younas, for his exceptional leadership, patience and mentoring. His guidance and support was unwavering throughout the period of this thesis. I could not have wished for a better supervisor.

I would also like to thank my co-supervisors Dr Arantza Aldea and Professor Hong Zhu, for their support, encouragement and insightful comments in the period of this research. I am truly grateful for such a wonderful team.

I will also like to thank my church family, members of Holding Forth the Word Ministry, Milton Keynes and in particular Revd. Biyi Ajala, for his support during the period of this study. I will also like to appreciate my first boss, Mr Isaac Orolugbagbe, for his support and for encouraging me to aim higher.

Also, my appreciation goes to my family members who have stayed by me the last few years and to my brothers Soji, Gbolahan and Bolaji.

I will particularly like to thank my father Mr Ayo Ogunyadeka for his immense support and also for bearing the entire financial burden of this research.

Finally, I will like to thank my wife Ibijoke for her patience and understanding during this period and to my wonderful son, Olufemi. God bless you both.

To God be all the Glory, great things He has done, greater things He will do!

**TABLE OF CONTENTS**

**Abstract**..... i

**Declaration** ..... ii

**Acknowledgements**..... iii

**TABLE OF CONTENTS** ..... iv

**LIST OF FIGURES**.....viii

**LIST OF TABLES**..... x

**LIST OF ABBREVIATIONS**..... xi

**CHAPTER 1** ..... 1

**INTRODUCTION**..... 1

**1.1 CLOUD COMPUTING** ..... 2

**1.2 NoSQL DATABASES and TRANSACTIONS**..... 3

**1.3 MOTIVATION AND RATIONALE OF THE RESEARCH** ..... 5

        1.3.1 Statement of the Research Problem..... 5

**1.4 AIM and OBJECTIVES** ..... 6

**1.5 RESEARCH METHODS**..... 7

**1.6 MAIN CONTRIBUTIONS**..... 8

**1.7 STRUCTURE OF THE THESIS** ..... 9

**CHAPTER 2** ..... 11

**BACKGROUND** ..... 11

**2.1 DATABASE MANAGEMENT SYSTEMS**..... 11

**2.2 DATABASE TRANSACTION MANAGEMENT**..... 12

        2.2.1 ACID Properties..... 13

        2.2.2 Serializability ..... 15

        2.2.3 Concurrency Control Techniques..... 16

**2.3 DISTRIBUTED TRANSACTION MANAGEMENT**..... 17

**2.4 TRANSACTION RECOVERY PROTOCOLS**..... 19

        2.4.1 Two Phase Commit ..... 19

**2.5 BIG DATA and NoSQL DATABASES**..... 24

**2.6 BIG DATA MANAGEMENT IN NOSQL DATABASES** ..... 25

## TABLE OF CONTENTS

2.6.1	Partitioning.....	25
2.6.2	Scaling .....	26
2.6.3	Replication .....	27
2.6.4	Failure Detection and Recovery .....	28
2.6.5	Load Balancing .....	28
2.6.6	Garbage collection .....	29
<b>2.7</b>	<b>ANALYSIS OF CAP THEOREM .....</b>	<b>29</b>
2.7.1	BASE .....	32
2.7.2	Other Consistency Models.....	34
<b>2.8</b>	<b>SUMMARY .....</b>	<b>34</b>
<b>CHAPTER 3</b> .....		<b>36</b>
<b>DATA PROCESSING IN CLOUD COMPUTING</b> .....		<b>36</b>
<b>3.1</b>	<b>ARCHITECTURAL CONSIDERATIONS of CLOUD DATABASES .....</b>	<b>37</b>
3.1.1	Loose Coupling VS Tight Coupling.....	37
3.1.2	Share Nothing VS Shared Disk.....	38
3.1.3	Data Model.....	38
3.1.4	Concurrency Control Techniques.....	39
3.1.5	Replication .....	39
3.1.6	Master-Slave VS Peer to Peer Architecture .....	40
3.1.7	Query Processing Approach.....	40
3.1.8	Read Optimised VS Write Optimised .....	41
3.1.9	Latency VS Durability .....	42
<b>3.2</b>	<b>NOSQL DATABASES AND BIG DATA .....</b>	<b>42</b>
3.2.1	BIGTABLE.....	44
3.2.2	MONGODB .....	45
3.2.3	DYNAMO .....	46
3.2.4	CASSANDRA.....	47
3.2.5	PNUTS.....	47
<b>3.3</b>	<b>TRANSACTIONS IN CLOUD DATABASES.....</b>	<b>48</b>
3.3.1	The Integrated Approach .....	49
3.3.2	The Middleware Approach .....	52
3.3.3	The API Approach.....	54
<b>3.4</b>	<b>ANALYSIS OF OTHER TRANSACTION MODELS AND PROTOCOLS .....</b>	<b>56</b>

## TABLE OF CONTENTS

<b>3.5</b>	<b>DISCUSSION AND CONCLUSION</b> .....	<b>58</b>
<b>CHAPTER 4</b> .....		<b>61</b>
<b>MODELLING AND DESIGN OF THE PROPOSED APPROACH – NoSQL-TX</b> .....		<b>61</b>
<b>4.1</b>	<b>NoSQL TRANSACTIONS</b> .....	<b>61</b>
<b>4.2</b>	<b>SYSTEM DESIGN APPROACH</b> .....	<b>64</b>
4.2.1	Snapshot Isolation.....	64
4.2.2	Rationale for Snapshot Isolation.....	66
<b>4.3</b>	<b>ARCHITECTURE OF THE NoSQL-TX</b> .....	<b>67</b>
4.3.1	Transaction Processing Engine (TPE) .....	67
4.3.2	Data Management Store.....	68
4.3.3	Time Stamp Manager.....	69
<b>4.4</b>	<b>TRANSACTION STATE TRANSITION MODEL</b> .....	<b>70</b>
<b>4.5</b>	<b>INTERACTION BETWEEN SYSTEM COMPONENTS</b> .....	<b>72</b>
<b>4.6</b>	<b>COMMIT PROTOCOL</b> .....	<b>75</b>
<b>4.7</b>	<b>ABORT SCENARIOS</b> .....	<b>77</b>
<b>4.8</b>	<b>A PROTOCOL FOR MANAGING TRANSACTIONS ACROSS</b> <b>ASYNCHRONOUS DATA REPLICATION</b> .....	<b>79</b>
<b>4.9</b>	<b>SUMMARY</b> .....	<b>82</b>
<b>CHAPTER 5</b> .....		<b>84</b>
<b>IMPLEMENTATION OF THE NoSQL-TX SYSTEM</b> .....		<b>84</b>
<b>5.1</b>	<b>DESIGN OBJECTIVES</b> .....	<b>84</b>
<b>5.2</b>	<b>IMPLEMENTATION TOOLS AND TECHNOLOGIES</b> .....	<b>85</b>
<b>5.3</b>	<b>IMPLEMENTATION OF TRANSACTION OPERATIONS</b> .....	<b>89</b>
5.3.1	Types of Operations .....	90
5.3.2	Aborts Scenarios for Operation .....	99
5.3.3	Optimisation Decisions .....	104
<b>5.4</b>	<b>APPLICATION DOMAIN</b> .....	<b>105</b>
<b>5.5</b>	<b>SUMMARY</b> .....	<b>109</b>
<b>CHAPTER 6</b> .....		<b>111</b>
<b>EXPERIMENTAL EVALUATION</b> .....		<b>111</b>
<b>6.1</b>	<b>EVALUATION BENCHMARKS AND WORKLOADS</b> .....	<b>111</b>
6.1.1	YCSB and YCSB+T Benchmark .....	112
6.1.2	Workloads for Experiments .....	112

## TABLE OF CONTENTS

<b>6.2</b>	<b>EXPERIMENTS AND RESULTS.....</b>	<b>116</b>
6.2.1	Experiment – Set 1 .....	118
6.2.2	Experiment – Set 2 .....	121
6.2.3	Experiment – Set 3:.....	122
6.2.4	Experiment – Set 4 .....	124
6.2.5	Experiment – Set 5 .....	126
6.2.6	Experiment – Set 6 .....	127
6.2.7	Experiment – Set 7 .....	128
6.2.8	Experiment – Set 8 .....	129
<b>6.3</b>	<b>ANALYSIS OF THE PROPOSED SYSTEM AND EXISTING APPROACHES</b>	<b>130</b>
<b>6.4</b>	<b>SUMMARY .....</b>	<b>132</b>
<b>CHAPTER 7</b> .....		<b>133</b>
<b>CONCLUSION AND FUTURE WORK</b> .....		<b>133</b>
<b>7.1</b>	<b>CONTRIBUTIONS .....</b>	<b>134</b>
<b>7.2</b>	<b>CRITICAL ANALYSIS .....</b>	<b>135</b>
<b>7.3</b>	<b>FUTURE WORK .....</b>	<b>136</b>
<b>REFERENCES</b> .....		<b>138</b>



# LIST OF FIGURES

Figure 2.1: Two Phase Commit State Diagram	21
Figure 2.2: CAP Theorem classification of NoSQL Databases	32
Figure 4.1: Snapshot Isolation	65
Figure 4.2: Transaction State Diagram	72
Figure 4.3: Component of Proposed System- NoSQL-TX	73
Figure 4.4: Interaction between Components of the System(NoSQL-TX)	75
Figure 5.1: MAAS Head Controller Configuration	87
Figure 5.2: Nodes in the cluster with their local addresses	88
Figure 5.3: Configuration of One of the Nodes - Address 10.0.0.110	88
Figure 5.4: MongoDB service running via Putty	89
Figure 5.5: Hardware Setup of Proposed System	89
Figure 5.6: Read Operation Codes in Python	91
Figure 5.7: Read-latest operation	93
Figure 5.8: Account Details for an Account User	106
Figure 5.9: Transaction Records	108
Figure 6.1: Transactions per Second	119
Figure 6.2: Workload F vs. Workload G	121
Figure 6.3: Average latency per transaction	122

## LIST OF FIGURES

Figure 6.4: Percentage of completed transactions	123
Figure 6.5: Percentage throughput of Multi-key transactions	123
Figure 6.6: Total Number of Aborted Transactions	124
Figure 6.7: Workload A vs Workload G (Aborted Transaction)	125
Figure 6.8: Period between transaction start time and commit time	126
Figure 6.9: Latency of Workload Requests	128
Figure 6.10: Transactional Overhead of Update Operations	128

# LIST OF TABLES

Table 2.1: Main Differences between NoSQL and Classical Databases	35
Table 6.1: Workloads for Evaluation	114
Table 6.2: Number of Clients and Operations Executed	116
Table 6.3: Total Number of Completed Transactions per Second	119
Table 6.4: Percentage distribution of Transaction	127
Table 6.5: Anomaly Score	129

# LIST OF ABBREVIATIONS

2PC:	Two Phase Commit
2PL:	Two Phase Lock
ACID:	Atomicity, Consistency, Isolation and Durability
ACP:	Atomic Commit Protocol
BASE:	Basically Available, Soft-state Eventual Consistency
CAP:	Consistency, Availability and Partition tolerance
DBMS:	Database Management System
DDBMS:	Distributed Database Management System
DMS:	Data Management Store
GFS:	Google File System
IaaS:	Infrastructure as a Service
JSON:	JavaScript Object Notation
MAAS:	Metal As A Service
MMDB:	Main Memory Database
NoSQL:	Not Only SQL
NoSQL-TX:	Not Only SQL Transaction
PaaS:	Platform as a Service
PNUTS:	Platform for Nimble Universal Table Storage
QoS:	Quality of Service
RDBMS:	Relational Database Management System
ROWA:	Read One Write All
SaaS:	Software as a Service
SOA:	Service Oriented Architecture
TPE:	Transaction Processing Engine
TSM:	Time Stamp Manager

# CHAPTER 1

## INTRODUCTION

Cloud computing technologies feature among the top ten most disruptive technological trends of this era [1]. Cloud computing offers a lot more flexibility than the traditional legacy systems and at a cheaper cost, making it more attractive to consumers as well as service providers.

Cloud service providers such as Google (Google Apps), Microsoft (Azure), Amazon (Amazon web services), and Salesforce (Salesforce CRM tools) provide infrastructure, platform and software services which are generally deployed (and run) on inexpensive commodity computing infrastructure. Such infrastructure is generally composed of tens of thousands of servers and network components which are located in different data centres around the world.

There has been rapid development in cloud and NoSQL (Not Only SQL) databases in order to store large volume of data and to make such data highly available and efficient for cloud service provisioning. However, there still exist important challenges such as security, privacy, network QoS, data consistency, availability, reliability, performance, environmental issues, economical and business related issues [2] [3] [4] that need further research. This thesis investigates into the transaction management of NoSQL databases in order to ensure consistency of cloud data and to provide appropriate level of reliability and performance.

This chapter is organised as follows. Section 1.1 describes background and fundamentals of cloud computing. Section 1.2 explains the characteristics of NoSQL databases and the properties of transactions. Section 1.3 explains the motivation and rationale for this research. It also specifies the scope of the research problem addressed in this thesis. Section 1.4 explains the aim and objectives of this research. Section 1.5 explains the research methodology.

## INTRODUCTION

Section 1.6 identifies the contributions of this research. Section 1.7 explains the structure of this thesis.

### 1.1 CLOUD COMPUTING

Cloud computing is defined as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [5] .

Cloud computing follows architectural designs and service provisioning models that deviate from the conventional perspectives, design and services of traditional information technology. Cloud computing offers information technology resources provisioned (mainly through the Internet) in a Service Oriented Architecture (SOA) [6] where users only pay for the resources they consume. Compared to classical Internet and distributed computing systems, cloud computing has various distinguishing features, including pay-as-you-go business model, virtualization, large scale storage facilities, support for big data, high compute power, and support for exploiting the strength of both powerful as well as commodity computers [7]. This model of pay-as-you-go services offered by cloud computing vendors gives consumers the illusion of access to infinite computing resources. Resources are completely elastic and can be provided for as and when needed. Organisations and individuals therefore do not need to make capital investments in computing resources. These features have empowered cloud computing technologies to become an attractive service hosting and delivery platform for various organisations, educational institutions, public sector and industries such as Google, Microsoft, Amazon and Facebook amongst others. Generally, cloud computing services are broadly categorized into three different types [2] including:

**Software-as-a-service (SaaS):** SaaS is a multi-tenant platform where users buy a subscription from the publisher or service provider. Publishers of software that

## INTRODUCTION

make use of SaaS have the advantage of deploying new features in their software as soon as they are released, while this may be very difficult to implement in traditional software. Salesforce is a major provider of SaaS.

**Platform-as-a-service (PaaS)** – PaaS provides developers with operating systems and environment (application development frameworks) comprising of end-to-end lifecycle of developing, testing, deploying and hosting of a web application as a service. Google (App Engine) and Microsoft (Azure platform) are major providers of this service.

**Infrastructure-as-a-service (IaaS)** – IaaS refers to on-demand provisioning of infrastructural resources, usually in terms of Virtual Machines. IaaS providers include Amazon (AWS), Dimension data and RackSpace.

As mentioned earlier, cloud service providers provide infrastructure, platform and software services which are deployed on inexpensive commodity computing infrastructure located in many data centres across the world. This level of service provisioning introduces challenges of security, consistency, availability and performance among other issues.

### 1.2 NoSQL DATABASES and TRANSACTIONS

Cloud service provisioning can handle large volume of data (referred to as Big Data) in order to support large scale applications and a large population of users/clients. For example, managing data related to millions of Facebook, Twitter or Google mail users needs effective data management and processing capabilities to achieve data consistency and high availability but in a reliable and efficient way.

NoSQL databases are most commonly used to process and manage Big Data which has distinguishing characteristics such as large volume, different variety and high velocity. NoSQL databases offer a paradigm shift from the complex query processing capabilities and monolithic architecture of the relational databases [8]

## INTRODUCTION

to simple queries but with parallel processing capabilities so as to increase performance.

However, this shift in paradigm and design has some negative effects on data consistency which makes NoSQL databases impractical for use in certain application domains that need strong consistency of data such as financial or banking applications. This is due to the fact that NoSQL systems do not offer support for transactions which is a classic feature of the relational databases.

Different models and techniques have been developed by different cloud computing and NoSQL database vendors in order to effectively manage the (large amount of) Big Data. Majority of the existing techniques focus on improving efficiency and availability of big data but do not give appropriate attention to ensuring data consistency. This research focuses on transaction processing in cloud computing and NoSQL databases with particular attention to the data consistency and the implementation of transactions in NoSQL databases.

## TRANSACTIONS

A transaction is defined as an execution of a software program which contains multiple 'read' and 'write or update' operations in order to read data from and update data in a database. In traditional databases, operations in transactions are concurrently executed in an interleaved fashion in order to make optimal use of computing resources [9]. Therefore, to ensure correctness and integrity of a database, transactions must obey the rules set by ACID (Atomicity, Consistency, Isolation and Durability) properties. Atomicity means all the operations in a transaction must be completely carried out otherwise none of the operations must be carried out. Consistency means after the completion of a transaction, the database must remain in a valid state. Isolation requires that transactions must not interfere with each other and durability ensures that changes made after a transaction remain permanent in a database [10].



### 1.3 MOTIVATION AND RATIONALE OF THE RESEARCH

In order for NoSQL databases to effectively scale data across multiple nodes, certain trade-offs have been made in their design [11]. These include:

- Simple data model which means data entities do not have to be normalized and can be spread across different nodes.
- No support for join or referential integrity which is easily achieved because of the simple data model.
- Relaxing the consistency guarantees and a lack of support for transactions

The above mentioned trade-offs have implications on the way NoSQL databases process and manage data. The simple model of the data means that there can be no complex operations or queries in NoSQL database. Most NoSQL databases have simple 'get' or 'put' requests and they do not support online transaction processing (OLTP) applications [12]. The lack of support for 'joins' and 'referential integrity' means that there can be no interaction between rows or tables in the database. Finally, managing consistency across replicated data is non-trivial. Replication could either be synchronous or asynchronous. In choosing synchronous replication, there is a risk of compromising data availability. If any of the replicas is not available, the data item is not available. On the other hand, with asynchronous replication, data consistency might be sacrificed. This means that a replica can be outdated and would still be allowed to respond requests (or read by applications). There are diverse adaptations of quorum based protocols [13] that are used to manage replicas such as paxos [14] [15], read one write all (ROWA) and read one write all available (ROWAA). Though there are various issues involved, the following section defines the research problem which is to be addressed in this thesis.

#### 1.3.1 Statement of the Research Problem

It is observed from the above discussion that NoSQL databases completely avoid the need to support multi-row operations. A multi-row operation is a group of operations affecting multiple key items (also referred to as multi-key). NoSQL

## INTRODUCTION

databases also do not implement ACID-based transactions. This decision is also supported by the fact that applications that make use of NoSQL databases such as Facebook, twitter, and emails do not necessarily need ACID level consistency [94]. Also, research shows that the rigidity of ACID transactions in relational database makes it unfit for certain large scale and complex applications [16].

Even so, enterprise users, whose applications are mainly driven by transaction processes, have shown little or no interest in NoSQL databases because of its lack of support for ACID transactions [17] [18]. Also, NoSQL databases are inadequate for future applications such as online gaming applications that make use of multi-user collaborations because such games need ACID transactional properties during the execution of a game. This has necessitated the need for research into how transactions can be implemented in NoSQL databases. Implementing transactions in NoSQL databases will definitely increase the functional applications of NoSQL databases which will in turn increase its use among enterprise users. Some of the existing research [19] also advocates the support of ACID transactions in NoSQL databases stating that it would not be wise to sacrifice the support for transactions and the “golden standard” ACID consistency of SQL databases. However, the major strength of NoSQL databases is their ability to scale seamlessly to a large volume of data by taking advantage of parallel processing. This research proposes an approach that combines the advantages of the two worlds i.e. taking advantage of the parallel processing power of NoSQL databases and at the same time not sacrificing the support for transactions and consistency in NoSQL databases.

### **1.4 AIM and OBJECTIVES**

The aim of this research is to investigate into the transaction processing and data management of NoSQL cloud databases in order to develop a new framework that enhances the efficiency, consistency and reliability of NoSQL cloud databases. To achieve this aim, the following objectives are defined:

## INTRODUCTION

- I. **Conduct an in-depth study of the literature and current state-of-the-art methods related to the data management and transactions in the cloud, including both NoSQL and classical SQL databases:** The objective is to identify current challenges and to explore potential research issues in cloud computing and NoSQL data processing. This will also explore ways in which the current methods used in transaction processing can be optimized in terms of efficiency, reliability and availability of cloud data.
- II. **Design a new framework for transaction management in NoSQL databases:** A new framework is to be designed which has the potential of addressing main research issues and to improve data consistency, reliability and efficiency in NoSQL databases.
- III. **Develop and implement the proposed framework as a prototype system using cloud data management tools and technologies:** The proposed framework is to be developed and implemented using cloud architecture and appropriate NoSQL database technologies.
- IV. **Test and evaluate the prototype system using cloud benchmarks:** Appropriate cloud testing benchmark will be followed in order to test the validity of the proposed framework in the transaction management of NoSQL cloud databases.

In order to achieve the above aim and objectives, this research will follow appropriate methodological approach, which is described in the following section.

### 1.5 RESEARCH METHODS

A research method is defined as all the techniques used during the course of a research to perform operations with the aim of providing a solution to the given research problem [20]. This research is carried out over a period of three (3) years and the approach followed which was adopted from a combination of methods [20] [21] [22] are outlined below.

**Define research problem** – The problem area is first identified and defined. From the definition of the problem, the aim and objectives of the research is clearly spelt out.

**Literature survey** – The approach to the literature review starts from a holistic study of the broader cloud computing technologies and service oriented architecture combined with a review of existing techniques of used in relational

## INTRODUCTION

database transactions. The study was then narrowed down to a study of big data management techniques, NoSQL databases as well as the CAP Theorem. Finally, this phase ends with an in-depth study of the state of the art techniques of implementing transactions in NoSQL database. From this study, the short-comings of existing systems and gap in knowledge are identified.

**Modelling and Specification** – having identified the short-comings of the current state-of-the-art systems, and with a proper understanding of the research area, this thesis proposes a model that aims to solve the problem of implementing transactions in NoSQL databases. Putting into consideration existing theorems and techniques, the thesis specifies the theoretical scope of the problem. This theoretical specification enables the components of the proposed transaction model to be formally verified [21].

**Develop Proposed Design** – Based on the specification of the model, the proposed design is implemented as a prototype using relevant tools of and techniques.

**Experimental Evaluation** – The Evaluation is carried out using benchmarks that are relevant to the transaction and cloud computing environments. The evaluation involved using a set of experiments which tests the performance and correctness of the system under load. The system is also compared with existing systems to evaluate its strengths and advantages [22]. From the Evaluation, the results are collected, analysed and interpreted to identify the strengths and weaknesses of the system.

### 1.6 MAIN CONTRIBUTIONS

This thesis designs and develops a new approach, called NoSQL-TX (NoSQL Transactions). The main contributions of NoSQL-TX approach are as follows.

1. An in-depth study of the architecture of NoSQL databases and clearer definitions of the factors that limit them from supporting transactions

## INTRODUCTION

2. The design and definition of a new Multi-Key transaction model for NoSQL databases that guarantee ACID properties of transactions in order to ensure stronger consistency and integrity of data.
3. The development of a loosely-coupled novel architecture that separates the implementation of transactional logic from the underlying data thus ensuring transparency and abstraction in cloud and NoSQL databases.
4. The design and implementation of a novel protocol for managing consistency across asynchronous replication in NoSQL databases.
5. The definition and implementation of new types of database operations ('read' and 'write') which provide flexibility for adjusting data consistency based on user requirements.
6. The development of a prototype system using real NoSQL system, MongoDB, which is evaluated using the YCSB+T benchmark based on standard Yahoo! Cloud Services Benchmark (YCSB). The proposed system is believed to enhance consistency and performance of NoSQL databases.

### 1.7 STRUCTURE OF THE THESIS

The rest of the thesis is structured as follows. :

- Chapter 2 reviews fundamental techniques in transactions and distributed transactions processing such as two phase commit and two phase locks. Furthermore, existing literature on the validity of CAP theorem and its implication on distributed databases is also critically analysed. The characteristics of big data and techniques used in managing big data are explained.
- Chapter 3 gives an overview of various NoSQL databases and their characteristics. A brief overview of various approaches to implementing transactions in NoSQL databases is given and finally, a review of state of the art NoSQL systems that support transaction is given with their shortcomings identified.
- Chapter 4 builds on top of the various system designs reviewed in chapter 3 and presents a theoretical modelling of the proposed solution. The architecture of the proposed system as well as the scope of the proposed NoSQL transaction are clearly explained and defined. The details of how

## INTRODUCTION

the system components interact to implement consistency are also explained.

- Chapter 5 explains the implementation details used to develop the prototype system. The application domain which is used to implement the system is explained along with the tools used in implementing the system. The algorithms which each of the operations follow during their execution are presented.
- Chapter 6 evaluates the proposed solution and the implemented prototype system. The justification behind the chosen workloads used to evaluate the system is also explained. From the evaluation results, the short-comings and trade-offs of the proposed system is deduced. The performance overhead of implementing transactions is clearly identified.
- Chapter 7 concludes this thesis with a summary of the contributions of this thesis and a critical analysis of the proposed system. The chapter also provides possible areas of improvement for the proposed system which can serve as direction for future research.

# CHAPTER 2

## BACKGROUND

Cloud based NoSQL databases and their transactional systems follow some of the principles and techniques of the classical (or standard) databases and transaction management. This chapter therefore reviews the fundamentals of classical database management systems in section 2.1 and explains database transactions in section 2.2. Section 2.3 explains transaction management in distributed databases and section 2.4 examines various transaction models and protocols. Section 2.5 describes the characteristics of big data and NoSQL databases while section 2.6 illustrates the techniques used in managing big data

Finally, section 2.7 describes and critically reviews the CAP theorem, which is the motivation behind the design of the NoSQL databases, and its implications on distributed system, cloud and NoSQL databases.

### 2.1 DATABASE MANAGEMENT SYSTEMS

A database management system (DBMS) is defined as “a software designed to assist in maintaining and utilizing large collections of data” [23].

A database is “a well-organized collection of data that are related in a meaningful way, which can be accessed in different logical orders” [24].

The objectives of using a DBMS are as follows [24].

**Data availability** – Database can be queried to retrieve information that would be meaningful to the user.

**Data integrity** – This means that the integrity of data must be preserved. The consistency and accuracy of data stored in a database must be maintained.

## BACKGROUND

**Data security** – Only users who have been granted authority may be allowed to access the data stored in a database.

**Data independence and transparency** – Users should not necessarily be concerned with how the data is physically represented on the database.

The most widely used DBMS is the Relational Database Management System (RDBMS) that was designed according to the above objectives. The RDBMS was designed based on strong mathematical principles [25] (set theory) that makes use of normalization and strong referential integrity in order to represent relationship among data entities. These characteristics, including the objectives stated above have influenced the way RDBMS support transaction management. However, as explained later, the relational databases are monolithic in their design and therefore cannot scale out to host large volume of data and a large population of users (as in cloud) [12].

## 2.2 DATABASE TRANSACTION MANAGEMENT

In database systems, one of the main strategies, to maintain the consistency of shared data during the concurrent execution of multiple requests (from multiple users), is the transactional management technique. Database systems group multiple read and write operations into (atomic) transactions that follow ACID (Atomicity, Consistency, Isolation, Durability) properties. In order to preserve data consistency, the execution of transactions must be serializable. A serializable execution of transactions is an execution whose output would yield the same result as when the transactions are executed serially. In relational databases, a scheduler is used to ensure that the executions of transactions are serializable. A scheduler is normally used in conjunction with a technique called locking [26]. Before a transaction starts, it acquires locks on all the data items involved in a transaction and holds the locks until all the operations in a transaction have been processed. During this period, no other transaction can change the data that has been locked. This would guarantee that transactions are isolated from each other. After processing the operations, a transaction releases all the locks. This process is



## BACKGROUND

called two phase locking (2PL). In the first phase, a transaction acquires locks for all the data items involved in a transaction. In the second phase, all acquired locks are released. After locks have been released, the transaction is not allowed to request for any other locks. This activity put in place to ensure that transactions would always leave the database in a consistent state, is known as concurrency control. However, the mechanism of concurrency control gets more complex in distributed database environment. The schedulers in each participating database are responsible for managing the subset of data which it stores. Distributed transaction is explained in [section 2.3](#). Essentially, cloud computing technologies fall into the category of distributed computing.

### 2.2.1 ACID Properties

The following scenarios describe two simple but crucial examples about databases and transactions. Consider a database that manages data of a banking application. Assume that the initial balance of a user X is £1,000 and the initial balance of a user Y is £1,000. Consider the following basic scenarios.

**SCENARIO 1: A user X intends to transfer an amount of money (say £100) from his account to user Y.**

User X initiates a transaction to carry out the transfer which should leave his account with a balance of £900 while user Y's balance should be £1,100.

The transaction begins by reading the initial balances of users X and Y. Assume that £100 was deducted user X's account but a failure occurred before that money was credited to user Y's account. This will leave the database in an inconsistent state. The database should have a way to react to this sort of failure and preserve the integrity of the data.

**SCENARIO 2: Two users A and B intend to transfer different amounts to the account of user X.**

Assume a user X has an initial account balance of £1,000. User A initiates a transaction  $T_a$  to transfer £5,000 to the account of user X and user B also initiates

## BACKGROUND

another transaction  $T_b$  to transfer £1000 to user X. These two transactions  $T_a$  and  $T_b$ , occur concurrently. Each transaction involves read and write operations.

After successful execution of the two transactions, the correct outcome is that user X should have a balance of £7,000. However, as stated in [section 1.2](#), operations in a computer system are carried out in an inter-leaved fashion i.e. they can be in any order so as to make substantial performance gains; as such the operations in scenario 2 can be carried out in the following order:

- |                                            |                              |
|--------------------------------------------|------------------------------|
| 1) Read <sub>a</sub> (Account X)           | returns £1000                |
| 2) Read <sub>b</sub> (Account X)           | returns £1000                |
| 3) Write <sub>a</sub> (Account X, + £5000) | balances account X at £6,000 |
| 4) Write <sub>b</sub> (Account X + £1000)  | balances account X at £2,000 |

Thus account X ends up with a balance of £2,000 instead of £7,000. If this scenario occurs, then  $T_a$  and  $T_b$  are said to be in conflict. Two transactions are in conflict if they concurrently operate on the same data item and at least one of them is a write [27]. Two transactions are said to be concurrent if their executions are overlapping, for instance, if the commit-time of one of them is in the interval between the start time and commit-time of the other transaction.

The above scenarios, though basic, forms the bedrock of transaction management in relational database systems. In order to preserve the consistency of a database, ACID properties of transactions were defined. The Atomicity property will prevent the scenario 1 from occurring because atomicity will mean that all operations of a transaction must occur or none of them. This would force a rollback in which the database is returned to its initial state before the transaction took place.

Database systems recognize certain key words that define the scope of the atomicity of operations in a transaction. These key words include:

**Begin** – This determines where the set of operations to be carried out as part of a transaction when it starts. Operations may be read or write operations.

## BACKGROUND

**Commit** – This is the point where the operations end and are committed atomically meaning that all operations between a ‘begin’ and a ‘commit’ must be successful. Failure of any operation in this scope would lead to a Rollback operation.

**Rollback** – The rollback operation is activated when there is a failure in any of the operations in a transaction. This means that all operations that have been executed must be undone to restore the database to its initial state.

The consistency property ensures that the database remains in a consistent state after the execution of transactions. For example, in scenario 1, the sum of both accounts should be the same before and after the transaction. The isolation property would prevent the occurrence of scenario 2. Isolation ensures that if two transactions are executed concurrently, the effect will be the same as if they are performed one after the other i.e. serially. For transactions to be isolated from each other, they have to be serializable. Durability would prevent loss of data and can be used for recovery from failures. In order to maintain these ACID properties, the DBMS employs certain techniques which are explained later in this chapter. Before going further, the concept of serializable execution of transactions and some of the anomalies caused by non-serializable execution of transactions are explained below.

### 2.2.2 Serializability

As explained earlier, a serializable execution of transactions is an execution whose output would yield the same result as when transactions are executed serially. Serializability is a technique used to preserve the isolation property of transactions. Without serialization, the executions of transactions are prone to certain errors identified below [28].

**Lost Updates** - Lost update occurs when two concurrently running transactions read a data and both of them write (or update) the same data. The effect of one transaction can cancel the effect of the other leading to a lost update. Scenario 2 (as above) demonstrates this error.

## BACKGROUND

**Inconsistent Reads** - An inconsistent read occurs when one transaction contains multiple write operations that modify (update) multiple data items. If this transaction has only executed some of the write operations and if another transaction reads the same set of data then it would see only a partial update of the initial transaction.

**Dirty Reads** - Dirty reads occur when a transaction makes changes to a data item and another transaction reads the changes made. The first transaction is then aborted and rolled back. This would mean that the second transaction has read a data item value that does not exist.

The next section explains certain concurrency control techniques implemented by DBMSs to achieve serializability and to maintain ACID properties in transaction execution of transactions.

### 2.2.3 Concurrency Control Techniques

Commonly used concurrency control techniques are as follows.

**Scheduling** – Database management systems have schedulers which manages the execution of transactions. The main function of a scheduler is to ensure that concurrent execution of transactions results in a serializable execution. To achieve this, when a scheduler receives an operation request, it can either execute the operation, delay or reject the operation depending on the state of concurrent transactions in a database. Schedulers help to reduce the possibility of conflicting operations. Schedulers make use of different techniques to guarantee isolation and consistency. Some of the main techniques include:

**Locking** – Locking implies that data items involved in an active (running) transaction are locked in order to prevent other transactions from accessing the same data items. This prevents transaction conflicts. Relational databases employs locking [26] to enforce isolation and consistency. Locking can also be referred to as pessimistic concurrency control mechanism. Most DBMSs use two phase locking [9] in order to implement locking in of data items. There are two variations of two phase locks namely conservative and strict. A conservative two phase lock requires that a transaction must obtain all the locks before it can

## BACKGROUND

proceed to carry out operations. On the other hand strict two phase locks requires that transaction do not need to retrieve all needed locks before it proceeds but locks held can only be released after the transaction aborts or commits.

**Write-ahead logging** - To enforce durability, DBMSs make use of a technique called logging. Logs are used for recovery purposes in the event of a failure [29]. The write-ahead log (WAL) protocol [30] [31] ensures that changes made to a data must be recorded on into a file on stable storage before the data is changed in memory. This ensures that in the event of a failure, the changes made to data can be replayed from the log file.

**Non – Locking Schedulers** – Aside from locking, other techniques such as Timestamp Ordering (TO), are used to ensure transaction isolation. In TO, each transaction is issued a unique timestamp. Transactions are then ordered according to their timestamps. If two transactions  $T_1$  and  $T_2$  conflict on a shared data item  $x$ , then operations in  $T_1$  must be processed before operations in  $T_2$  if and only if transaction  $T_1$  receives its unique timestamp before transaction  $T_2$ .

### 2.3 DISTRIBUTED TRANSACTION MANAGEMENT

Distributed transaction management has been conventionally implemented in distributed databases. According to [32], *“a distributed database is a collection of multiple, logically interrelated databases distributed over a computer network”*. A distributed database management system (DDBMS) is also defined in [32] as *“a software system that permits the management of the distributed database and makes the distribution transparent to the users”*.

The process of enforcing ACID across transactions in distributed database management systems (DDBMS) adds more complexity to concurrency control mechanisms than non-distributed databases. In distributed databases, the users are abstracted from the complexities of interactions among computers. To achieve this, it is important to create standards on which computer interactions

## BACKGROUND

are based. The keywords identified in [section 2.2](#) are often wrapped up in remote procedure calls [33] (RPC) in what is known as Transactional RPC<sup>1</sup> [34] and are used as a method for computer interactions. Various protocols are used to implement ACID transactions across in distributed database systems. Some of the techniques highlighted in [section 2.2.2](#) are also applied in distributed transactions howbeit somewhat modified. Below is a brief explanation of the techniques.

**Distributed Two Phase Locking** – As stated earlier, locking ensures that conflicting transactions that are executed serially. In a single stand-alone DBMS, locking is implemented by a protocol called two phase locking (2PL) earlier discussed. However, the mechanism for implementing two phase locking is more complicated in a distributed DBMS. Each DBMS has a local scheduler that is responsible for managing data stored on it. When a distributed transaction is initiated, the transaction sends its operations (read and write) to each DBMS involved in the transaction. The local scheduler of each DBMS then allocates locks for each of the data item stored in its local site. The lock information is sent to the scheduler of all the participating sites.

**Distributed Timestamp Ordering** - In this technique, each site (component database) also has its own schedulers that issues timestamps to transactions. The decisions taken on each transaction is entirely left to the scheduler of that system

**Deadlock Management** – A deadlock occurs when two transactions are waiting for each other to release their locks on data items. Consider a scenario where two transactions  $T_a$  and  $T_b$  are concurrently running and both involves two data items X and Y.  $T_a$  locks data item X and  $T_b$  locks data item Y. In this situation,  $T_a$  would have to wait for  $T_b$  to commit so it can obtain lock on Y while  $T_b$  would have to wait for  $T_a$  to commit before  $T_b$  can obtain a lock on X. This leads to a deadlock since both transactions are waiting for each other. One way to detect deadlocks is to use timeouts.

---

<sup>1</sup> Transactional RPCs page 21

## 2.4 TRANSACTION RECOVERY PROTOCOLS

Recovery is defined as “the activity of ensuring that software and hard-ware failures do not corrupt persistent data” [9]. The Atomic Commit Protocol (ACP) is a procedure that ensures that all participants involved in a transaction either commits or aborts that transaction in their local sites. In other words, it guarantees that all the participants in a transaction reach the same decision to preserve data integrity. This is very important because in distributed environments, any of the participating sites can fail. It is important that on recovery, a failed site must reach the same decision as the other sites. The ACP which are discussed in the next sections guarantees the following criteria [9]:

- All sites must reach the same decision
- Once a decision has been made, it cannot be changed
- The decision to commit can only stand if all sites agree

There exist various ACPs such as two-phase commit, three-phase commit, presume abort, presumed commit and so on [34]. In the following, two-phase commit is explained given that it is a widely used protocol. However, detailed description and analysis of such protocols are beyond the scope of this thesis.

### 2.4.1 Two Phase Commit

The two phase commit protocol (2PC) is a protocol used to guarantee ACID consistency in a distributed database as well as web-databases. The 2PC aims to achieve a form of consensus among participating systems. In 2PC [35]

- Each site has the responsibility of logging the actions that takes place at that site. There is no notion of a global log.
- Exactly one site must play the special role of coordinator which is usually the site where the transaction originates. The coordinator site makes the final decision on whether the transaction should commit or abort.

## BACKGROUND

- Messages are exchanged between the coordinator and the other participants. Each participant logs the message that it sends out to help it recover from a failure.

Under normal circumstances when there is no failure, the protocol is completed in the two phases stated:

**Phase 1:** This phase, also known as the *voting* phase, involves the following steps:

- The coordinator sends a message (vote request) to each of the participant asking if they are ready to vote. After sending this message, the coordinator enters a state known as a **WAIT** stage. This message is also logged at the coordinator site.
- Once the message has been sent, each site that receives the message responds to the coordinator with a YES or NO message after the individual decisions have been logged at each individual site. If any of the participants decides a NO, that participant automatically aborts the transaction otherwise the participants enter a **READY** state.

**Phase 2:** Phase 2 which is called the commit phase or decision phase is carried out as follows.

- The coordinator receives the decision made by all the participants involved in the transaction.
- If all the participants responded with a YES vote, the coordinator sends out a message instructing all the participants to commit the changes made by the transaction and enters a **COMMIT** state. If one or any of the participants responded with a NO message, then the coordinator instructs all participants to abort and enters a state of **ABORT**.
- Each of the participants that voted YES awaits the decision from the coordinator. If they receive a commit message from the coordinator, they commit the operations at their local site and responds with an acknowledgement to the coordinator. The coordinator then completes the transaction. If on the other hand they receive an abort message from the coordinator, they begin a rollback of all the operations performed and respond with an acknowledgement to the coordinator.



## BACKGROUND

As stated earlier, each of these steps taken by any of the participants are logged locally at the participant site. A participant is in a period of **uncertainty** when it responds with a vote YES to the coordinator and is yet to receive an instruction from the coordinator in other words when it is in the **READY** state. See Figure 2.1.

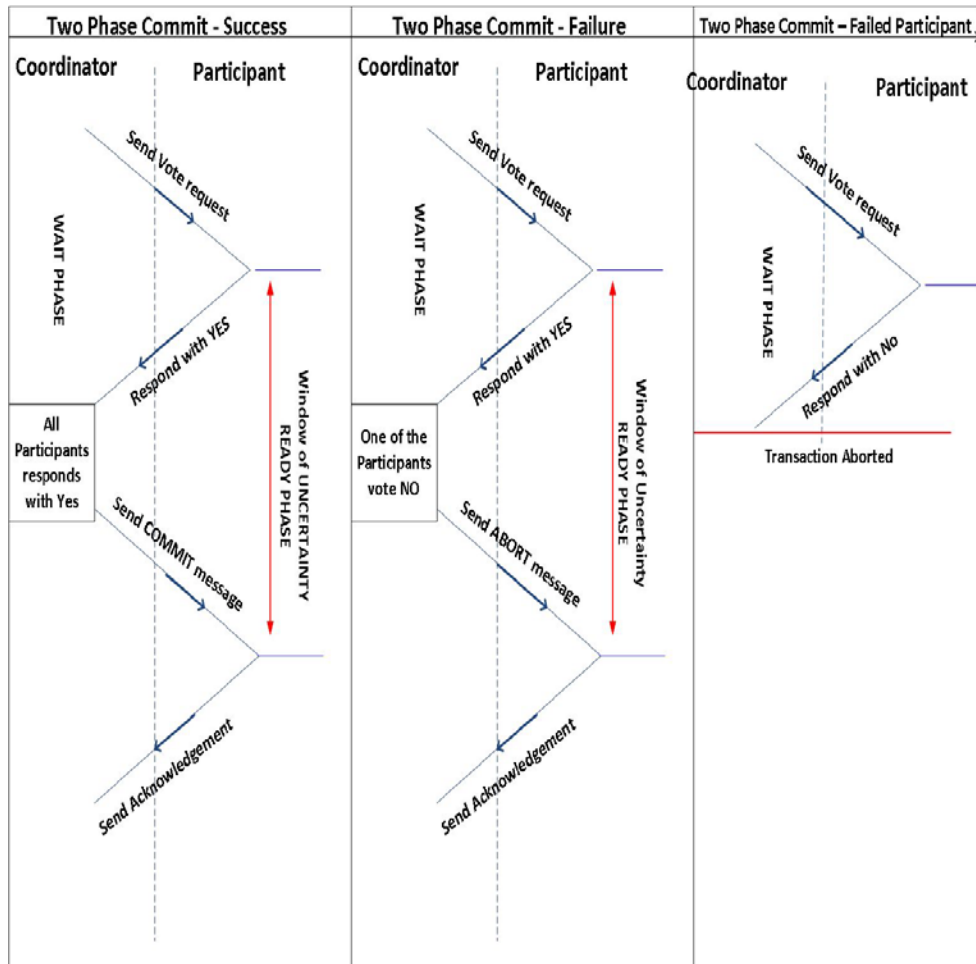


Figure 2.1: Two Phase Commit State Diagram

During the execution of a 2PC protocols, failures can occur. Such failure can be as a result of the following

- 1) Loss of messages as a result of network failures
- 2) Duplication of messages
- 3) Failure of any of the servers involved in the protocol

During a 2PC protocol, the participants and coordinator have to wait for messages. To prevent a transaction from unnecessary delays, timeouts actions are introduced. In phase one, the participants wait for a *vote-request* from the coordinator. Also, the coordinator has to wait for a response from the participant

## BACKGROUND

after sending a *vote-request* (i.e. during its *WAIT state*). In the second phase, participants that voted YES after receiving the *vote-request* from the coordinator also have to wait for a *commit* or *abort* message from the coordinator (i.e. when they are in a *READY state*). At this point, the participants are said to be in an ***uncertain*** state. This is because the participant has voted a YES to commit but cannot take a decision to commit or abort until it waits for the coordinator. However, if the participant had voted NO, the participant can as well proceed to abort the transaction on its site knowing fully well that the transaction cannot proceed on other sites because it has voted a NO. Also, according to the rules that guide an ACP, once a participant has made a voted to commit, it cannot change its mind. If a participant at this point is unable to reach the coordinator, the *restart* and *termination* protocols determine the behaviour of a system after a failure or timeout. A *restart* protocol specifies how a protocol should be restarted in event of a failure. A *termination* protocol specifies the procedure a transaction should follow in event of a time out. The *termination* and *restart protocol* has several implementations depending on the state of the transaction when time-out occurs. The following are the various implementations of the protocol [36]

**Coordinator Restart Protocol** - specifies how a coordinator should restart in the events of its failure. If a coordinator fails either before sending a *vote-request* message or at a *WAIT state* and cannot recall the responses of the participant, it can either re-send the request or abort the transaction. If it however fails after sending a commit or abort message, it must send the commit or aborted message again (as the case may be) and wait for the participants response.

**Coordinator Termination Protocol** - specifies how the coordinator should behave in the event of a time out due to failure of a participant or loss of message from a participant. In the event of a time-out, the coordinator resends the message according to the current state of the transaction.

**Participant Restart Protocol** – Specifies how a participant should proceed after it recovers from a failure. If a participant fails before receiving a *vote-request*, it can decide to respond with a **NO** when it receives the request. If the failure occurs during the prepared state, in other words, after it has responded with a **YES**

## BACKGROUND

message (Uncertainty period), it must wait for the coordinator to resend the commit or abort message.

In particular cases, where a participant cannot establish a connection with the coordinator, it can be allowed to contact other participants to know the final decision of the coordinator. The participant in this scenario, also known as the *initiator*, sends a *Decision-request* message to some other participant, known as the *responder*. If the responder has received a commit or abort message from the coordinator, then the participant commits or aborts as the case may be. If the responder has not voted, then it can decide to vote a **NO** and respond to the initiator with a NO message. However, if the responder has voted **YES** but has not received a commit or abort message from the coordinator, then the initiator would have to wait in this uncertain state. This protocol is referred to as the ***Cooperative Termination Protocol***.

**Participant Termination Protocol** - specifies how a participant should terminate in the event of a coordinator failure. If a participant times out while waiting a *vote-request*, it can decide to abort and respond with a **NO** when it receives the request. Its behaviour in other scenarios is essentially the same as the participant restart protocol.

Therefore, from the above, when a participant is in a state of uncertainty and it is unable to contact other participants or the coordinator, it is said to be in a *blocked* state. To prevent transactions from entering a blocked state, the three phase commit protocol was proposed.

Similar to 2PC, other protocols (such as 3PC, presumed abort, presumed commit, etc) ensure that database systems in distributed environment reach an agreement during concurrent implementation of transactions in order to preserve the consistency of data and correctness of applications.

## **SUMMARY OF (CONVENTIONAL) DATABASES AND TRANSACTIONS**

The above sections provided an overview of the conventional databases and related models and protocols of transaction management. Such databases have been dependent on relational database model. Relational databases are predominant for storing structured data and following ACID properties in order to

## BACKGROUND

maintain consistency of data. They also support relationships, associations and normalization of data. With all such features relational databases have proved very useful for applications that require strong consistency such as banking applications, e-commerce and online shopping, etc.

Despite the above benefits, relational databases are no longer sufficient for the needs of modern (Internet and Cloud) applications such as social media, business analytics, and online reviews. The speed and scale at which data is generated/processed is beyond the processing capabilities of the relational databases and transaction management techniques. These applications (or services) generally do not need well-structured and normalized data nor do they need strict consistency and ACID style transactions. They demand new ways of managing data and transactions in an easy and efficient manner. This has resulted into the new theories, techniques and technologies such as NoSQL databases and big data, CAP theorem, BASE properties, etc.

### **2.5 BIG DATA and NoSQL DATABASES**

Big data is characterised by 3Vs (Volume, Velocity and Variety) [37][38] or 4Vs (Volume, Variety, Velocity, and Value) model. Volume refers to large size of data that is possibly beyond the processing capabilities of conventional database; Variety means that big data may have structured, semi-structured and unstructured formats. Velocity indicates the speed at which data is generated which is usually high; Value refers to benefits that can be derived from the data.

Various techniques are employed to manage Big data efficiently. These techniques are explained in [section 2.6](#). Big Data is generated on daily basis from a number of sources which include data warehouses, sensor networks, text search, scientific databases and XML databases [39]. Commerce and business, society administration and scientific research are three identified areas that currently produce and make use of big data and data intensive applications [40]. Also, the increase in the use of online services has led to the design of a variety of web applications (e.g., social media, road traffic, etc) that generate a large volume

## BACKGROUND

of data. However, traditional databases are inappropriate to meet the demands of such applications [41], for example processing data of millions of tweets in real time. The concept of “One size fits all” in the database industry is no longer sufficient [39]. This has led to the design of specialized databases called NoSQL databases which are well explained in the next chapter. Processing Big data have certain requirements that are lacking in the traditional database. These requirements include elasticity, scalability, flexibility and fault tolerance [42] [43]. The fault tolerance feature prevents any single point of failure across the system. As stated earlier, failure is a norm in these environments and should not prevent the smooth running of the database. Even so, the difference between elasticity feature and scalability is emphasized. Elasticity requires that the system should be able to scale up or scale down as the need requires while scalability refers to the ability for a database to be scale across multiple systems. The 3/4Vs characteristics of Big data make NoSQL databases better suited for processing big data. NoSQL databases are have simple data models and can process unstructured data without the need for normalization [44]. However, most NoSQL databases can only perform simple operations and single key transactions. The various NoSQL databases and their characteristics are explained in the next chapter. However, below is a brief explanation of some of the main techniques employed in NoSQL systems to process big data.

## 2.6 BIG DATA MANAGEMENT IN NOSQL DATABASES

There are various techniques involved in managing big data in the cloud environment. Some of the commonly used techniques are explained below

### 2.6.1 Partitioning

Partitioning involves splitting a database into smaller parts called partitions [7]. There are two types of partitions namely: Vertical partitions and Horizontal partition. In vertical partitioning, a database table is split along the column attributes while in horizontal partitioning, the database is split by the rows. When

## BACKGROUND

a single database is partitioned and split (or scaled) across multiple servers, each server that manages a section of the database is known as a database shards [45]. This process is known as sharding. Sharding is a form of partitioning and both partitioning and sharding involves splitting the database into partitions (or sections). In partitioning, each section of the split database may or may not be managed by a separate server. However, in sharding, each section is managed by a separate server. There are three types of partitioning namely Hash partitioning, Range partitioning and Robin-round partitioning [46].

**Hash partitioning** – To implement hashing partitioning, a hash function is applied on each data key. The output of the hash function would determine the node that hosts the key. Hash partitioning is more suitable for applications that make extensive use of random scans. To find any data item, the hash function is applied to the key of the data. The result would yield the location of the data item.

**Range partitioning** – In range partitioning, each node stores a distinct range of data keys. Range partitions works efficiently with applications that mainly need sequential scan as most data items whose keys are closely related will be stored on the same node.

**Round-Robin Partition** – In round robin partitioning, key items are distributed evenly (in a ring fashion) according to the number of nodes. For instance, if there are 3 nodes, key items 1 to 3 will spread across node 1 to 3, key items 4 to 6 will spread across node 1 to 3 and so on.

### 2.6.2 Scaling

Scaling is a technique used to increase the processing capability of a database node and is of two types namely vertical scaling and horizontal scaling. In vertical scaling, the number of processors, memory size and disk size of a machine is increased to enable the machine process more data. Horizontal scaling on the other hand means adding more machines or increasing the number of nodes involved in processing data. The rationale behind horizontal scaling is in two dimensions. Firstly, doubling the hardware will reduce the time taken to perform a task by half and secondly, doubling the hardware will perform twice as much

## BACKGROUND

task in the same time [46]. In vertical scaling, there is limitation to how much a single node can scale vertically [47]. Also, to make a case for parallel processing (horizontal scaling), recent research [48] has shown that Moores law (which states that “the number of transistors on a microprocessor chip will double about every two years”), is fast becoming unachievable. This makes horizontal scaling more practical in big data processing than vertical scaling. NoSQL databases are designed to scale horizontally while relational databases are designed to scale vertically. This gives the NoSQL databases a greater advantage as there is no limit to the number of nodes that can be added to the database cluster.

### 2.6.3 Replication

Replication is a process of maintaining multiple copies of a database in different locations to provide fault tolerance. This results in higher levels of availability but introduces a new set of challenges. Replication is classified into two types namely **Eager** (synchronous) and **Lazy** (asynchronous) replications [49]. In eager replication, replicas are updated during the transaction while in lazy replication, replicas are updated at a later time. Keeping the replicas consistent introduces a set of problems and may involve certain trade-offs. Eager replication reduces performance, involves higher bandwidth and increases latency of transactions. Lazy replication on the other hand, means that data on some replicas can be stale and out-of-date. Out-of-date replicas may not be allowed to respond to client requests in applications that need high level of data consistency. The number of replicas implemented by a database system also has contradicting effect on the system. For instance, a high number of replicas imply that the system is able to provide stronger tolerance to fault. However, a high number of replicas also imply that more effort (in terms of bandwidth and latency) will be needed to keep the replicas consistent. There are various protocols and techniques used by developers to optimize their replication processes. One of such techniques includes primary – secondary replication where secondary replicas can process reads while only a primary replica can process writes (or updates). Also there are various quorum or consensus protocols used to guarantee consistency across replicas. The replication model adopted is dependent on application specific needs. All NoSQL databases make use of some form of replication for fault

## BACKGROUND

tolerance. Dynamo [50] uses consistent hashing for replica placement and eventual consistency model for replica management. Google Megastore [51] uses Paxos [15] to synchronously manage writes across replica.

### **2.6.4 Failure Detection and Recovery**

As explained earlier, in cloud computing environment, failure is a norm. This is because hundreds to thousands of machines are used to process data in parallel and these machines are mainly commodity machines. There has to be an efficient mechanism to detect machines that have failed in order to bring them up to date to guarantee availability and consistency. Most clusters have algorithms for detecting failure. Dynamo uses gossip based protocol (explained later in [section 3.2.3](#)). BigTable uses heartbeat messages which are exchanged between master server and slave servers. If there is no response from a particular node within a set timeout period, that node is assumed to have failed. Each of these systems implements various techniques for recovering a failed node after detecting its failures.

### **2.6.5 Load Balancing**

Load balancing [52] is a technique used to manage, distribute and re-distribute data across nodes to ensure that no single node is overloaded. The objectives of load balancing and distribution are to achieve high throughput, efficient resource utilization, low latency and to avoid hotspots across the cluster. Load balancing also aims to improve fault tolerance and to optimize the process of replica distribution in a cluster system. For instance, shard aware or rack awareness is a load balancing technique that ensures that replicas are distributed in such a way that network failures on a single rack do not affect availability. A common known rack aware technique is that used in HDFS [53] where two replicas are placed on two different nodes in a local rack while the third replica is placed on a different node in a different rack. That way, a network partition to a rack will not affect the availability of any data item.



## BACKGROUND

### 2.6.6 Garbage collection

Most cloud databases tend to keep more than one version of all data items to guarantee availability even at the expense of consistency. There is need to have an effective garbage collection process to effectively manage computing resources (storage/memory) and to prevent the database from storing unnecessary / unused data. Also, garbage collection of log files must be carried out with care to ensure that only logs that would not be needed are deleted. The process of garbage collection should not affect the smooth running of the system. In [54], garbage collection is carried out in batch when the master is in a quiescent state.

The above techniques are used in various degrees when managing big data. The overall combination of these techniques implemented by a database management system determines how suitable that database is for any application.

## 2.7 ANALYSIS OF CAP THEOREM

The CAP Theorem [55] states that a distributed system can offer at most two of the three desirable properties, Consistency, Availability and tolerance to network Partition (CAP). The cloud computing environment is characteristically a distributed environment and therefore the NoSQL databases are built to scale across multiple nodes. Techniques such as partitioning, horizontal scaling and replication are used to achieve high availability which is one of the unique characteristic of NoSQL databases. NoSQL databases cannot provide the three afore-mentioned properties simultaneously and are designed to trade-off either one of consistency and availability. However, the implication of the CAP theorem is that if the high availability characteristic of the NoSQL databases is to be guaranteed, consistency must be sacrificed. However, consistency is an important feature of transactions in relational databases. The CAP Theorem thus implies that NoSQL systems cannot support ACID level consistent transactions.

## BACKGROUND

Three fundamental requirements to provide scalable network services needed in cloud computing are identified [56] . They include

- 1 incremental scalability and overflow growth provisioning
- 2 “24/7 availability through fault masking” and
- 3 Cost effectiveness

The three points to note in these requirements include:

**Scalability** - The ability for resource provisioning to be increased as user needs increases.

**Availability** - The promise of uninterrupted access to resources

**Cost Effectiveness** - Services must be economically justifiable.

Cloud computing services promise high availability and the make-belief that computing resources are inexhaustible and available on demand. In order to provide high availability, replication is needed. Cloud service providers / vendors make use of highly distributed systems, replicated on a global scale. This normally would involve the use of hundreds or even thousands of machines and in these environments, failures are not a rare occurrence. These systems aim to achieve high availability, low latency, partition-tolerance and high scalability.

In distributed systems, enforcing ACID properties require substantial effort. Moreover, achieving ACID level guarantees in a distributed environment where data is replicated over large geographical area is a highly demanding and non-trivial task. Some of these systems are expected to handle a very high write throughput, billions of writes per day and are also expected to scale with the number of users [57]. In the presence of these failures, availability must be preserved because it is a key component of cloud computing as consumers must have access to computing resources. Implementing complex techniques like 2PC in cloud environment can be counter-productive. The cost of resolving the conflict between data consistency, system state and high availability is made more complex by the magnitude and robustness requirements of present day applications used by businesses.

## BACKGROUND

However, the ability to be able to scale data comes with its own challenges. Distributed environment face the problems of network reliability, system failures, network security and latency. If there is a partition in the network, data servers may become unavailable. All these led to the formulation of the CAP Theorem which has been proven formally [58]. However, there have been some questions raised about the validity of the CAP Theorem. The variance in the meaning of the *consistency* in ACID and the *consistency* in the CAP theorem also led to some misconceptions. Consistency in ACID means that a transaction leaves the database in a consistent state and obeys all integrity rules. On the other hand, consistency in CAP refers to maintaining a single copy consistency across replicas. In [59], the author argues that CAP is confusing because it implies that systems are restricted to only two of the three properties. This can imply that distributed systems cannot be available and consistent at the same time, a situation which is clearly impractical. The timeline consistency model of PNUTS [60] (a NoSQL database designed by Yahoo), provides the basis for this argument. PNUTS relaxes its consistency and only guarantees that updates on replicas will be applied in the same order at all replicas but does not guarantee that all replicas will be up-to-date. Also, if the master replica for a particular data item is unavailable, then that data item is altogether not reachable. This would imply that the system gives up or relaxes both availability and consistency. The choice of relaxing consistency across replicas was as a result of the cost of implementing synchronous updates over a wide area network. PNUTS rather reduces the latency of updates by implementing an asynchronous model of replication. This model ensures that application developers do not need to worry much about implementing consistency. However, as can be seen, consistency is still relaxed. A model called PACELC has been suggested as a substitute for the CAP Theorem [61] [12]. PACELC means that “if there is a partition (P), how does the system trade-off between availability and consistency (A and C); else (E) when the system is running as normal in the absence of partitions, how does the system trade-off between latency (L) and consistency (C)?” [61]. This argument was also put forward in [62] with the author stating that “the CAP theorem only prevents everybody from being consistent and available”. This seems to imply that even when there is no partition, distributed databases cannot experience both consistency and availability at the same time. There have also been other arguments about the

## BACKGROUND

confusion caused by the CAP theorem [63], [64] which was acknowledged by the proponent of the CAP theorem [65]. However, the CAP theorem was the basis for the design of the NoSQL databases and justifying the semantics of the grammar used in the CAP theorem is not the focus of this work. With this understanding, the classification of NoSQL databases according to their characteristics on the CAP spectrum is described below.

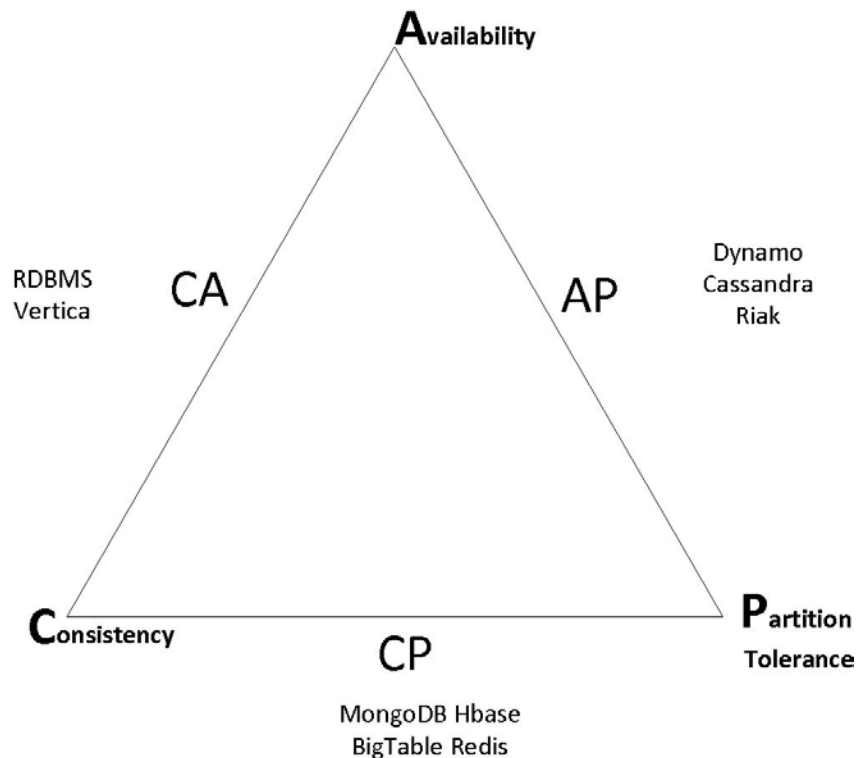


Figure 2.2: CAP Theorem classification of NoSQL Databases

As an outcome of the CAP theorem, various other models of consistencies were proposed. BASE as a consistency model for distributed systems was proposed by the proponent of the CAP theorem and is discussed in the next section.

### 2.7.1 BASE

The Basically Available Soft-state Eventual consistency model (BASE), suggested by Brewer, guarantees that after a specific time, all replicas would have received the update [66]. In BASE, the consistency part of the ACID properties of transactions is deliberately relaxed. Replicas do not necessarily need to have a consistent view of a data item. To guarantee that after a specific time, replicas

## BACKGROUND

would have received all updates, there is usually exchange of messages between the replicas. There are various protocols used to implement this exchange of messages. These protocols are discussed in later sections. The NoSQL databases adopt this model of consistency. Under the BASE, various levels of consistency guarantees exist. The following are examples illustrating the different consistency levels that can be applied [67].

**Strong consistency** – After an update, all replicas immediately return updated value. This provides ACID level consistency guarantees. It is usually achieved by ensuring synchronous replication.

**Weak consistency** – The system does not guarantee that subsequent access will return updated value until a number of conditions are met which could be a time frame in which all replicas would have been updated. These systems offer BASE level guarantee.

**Eventual consistency** – This is a form of weak consistency. In the absence of failures, the maximum time allowed for replicas to have inconsistent values can be determined based on certain factors like number of replicas. In this work, a different type of consistency model which follows the eventual consistency model (for asynchronous replication), but provides a stronger consistency guarantee for operations is proposed.

In [67], a quorum model for measuring the trade-of between availability and consistency among replicas is defined and explained. The following definitions hold:

N = No of replicas

W = Number of replicas needed to accept writes

R = Number of replicas needed to reach a read quorum

A distributed storage systems that need high availability and performance uses the configuration  $N = 3$  ( $W=2$ ,  $R=2$ ). For such systems, a read quorum would always overlap with write quorums thus guaranteeing a stronger consistency. However, for systems that serve very high reads and want to ensure a high tolerance to partitions and high availability, N can be as high as 10 while R can be as low as 1. This way availability is always guaranteed but consistency is very

## BACKGROUND

weak. If  $R=1$  and  $N=W$ , the system is highly optimized for reads, and if  $W=1$  and  $R=N$ , the system optimize for a very fast write. In summary, for most eventual consistent databases, the configuration used is  $W+R \leq N$ . This configuration would imply that read and write sets do not necessarily overlap.

### 2.7.2 Other Consistency Models

There are various other models of consistency that has been suggested in literature. A consistency model that allows users to determine the level of transaction consistency is proposed in [68]. Users are provided with three levels of consistency from which they can choose. Users are also not restricted to a particular consistency guarantee and they can switch their consistency guarantees at runtime depending on their needs. In timeline consistency, [60] the order of updates is preserved on all replicas, albeit asynchronously.

## 2.8 SUMMARY

Though conventional databases and transaction management techniques are different, they do provide the basis for the cloud and NoSQL databases and their transaction management.

This chapter therefore discussed the fundamentals of database systems and transaction management techniques and protocols. The chapter also explained how transactional properties are guaranteed in distributed database systems particularly focussing on the two phase commit protocol (used to guarantee transaction recovery). The characteristics of Big data as well as the techniques used to manage big data were also discussed. Finally, an analysis of the CAP theorem was explained.

Table 1 summarises the main differences in techniques employed by NoSQL cloud databases and classical relational databases.

Table 2.1: Main Differences between NoSQL and Classical Databases

<b>FEATURES</b>	<b>NOSQL DATABASES</b>	<b>CLASSICAL RELATIONAL DATABASES</b>
<b>CONSISTENCY</b>	Eventual Consistency	Strong consistency
<b>SCALABILITY</b>	Horizontally scalable	Vertically scalable
<b>REPLICATION</b>	Multiple replicas	Primary - Secondary replication
<b>AVAILABILITY</b>	Always available	Can be unavailable during upgrades
<b>TARGET APPLICATION</b>	Social networking, emails, big data	Banking applications, OLTP applications
<b>DATA MODEL</b>	Unstructured data with dynamic schema	Structured Data with defined schema and referential integrity between tables
<b>QUERY LANGUAGE</b>	Varies with databases (mainly programming languages)	Structured Query Language (SQL)
<b>TRANSACTION PROCESSING</b>	No Transactions	YES- ACID Transactions
<b>QUERY COMPLEXITY</b>	Simple 'get' and 'put' operations. No Support for table joins	Support complex query and supports table joins
<b>TOLERANCE TO PARTITION</b>	YES	NO
<b>ARCHITECTURE</b>	Loosely Coupled	Tightly coupled - Monolithic

The next chapter reviews related work on cloud and NoSQL databases and transaction management techniques and protocols.

# CHAPTER 3

## DATA PROCESSING IN CLOUD COMPUTING

One of the key characteristics of cloud computing is the make-belief to consumers that computing resources are inexhaustible and available on demand [69]. Providing these kinds of services involve the use of hundreds or even thousands of commodity machines. In these environments, failures are not a rare occurrence as some of these systems are expected to handle a very high write throughput such as billions of writes per day. They are also expected to scale with the number of users. In the presence of these failures, availability must be preserved because it is a key component of cloud computing as consumers must have access to computing resources and data. To guarantee availability, these systems make use of techniques such as partitioning and replication. As described in Chapter two, traditional databases do not scale due to the complexity of their data model [70]. The characteristic of a database that allows it to scale across multiple systems is called scalability. NoSQL databases are highly scalable and as such, a perfect fit for cloud environment. They are designed to scale up and scale down as and when the need arises. However, for a database to be scalable, certain design trade-offs are made.

This chapter therefore examines in section 3.1 the various architectural designs that cloud database vendors consider when designing their database. Section 3.2 gives examples of how these designs are implemented in some of the popular NoSQL databases highlighting their strengths and weaknesses. Furthermore, section 3.3 explains the state of the art approaches used in implementing transactions in NoSQL databases. Finally, section 3.4 gives a comparative analysis of the existing implementations and identifies the main research issues which are to be addressed in this thesis.



### **3.1 ARCHITECTURAL CONSIDERATIONS of CLOUD DATABASES**

In designing a cloud or NoSQL database (in this thesis, 'cloud databases and NoSQL databases are used interchangeably), certain architectural considerations are to be put into perspective based on the requirements of an application which is to be hosted in a cloud. Different cloud vendors have come up with different decisions about cloud set-up, databases and related cloud applications. Thus, the various design decisions taken by cloud vendors on the operations of their systems have implications on the characteristics and properties of their databases and the kind of applications that can be managed by such databases. Also, the design decisions taken by the vendors determine factors such as the consistency of the system and the types of operation the database can handle. This will have an impact on the ability of the database to be able to manage transactions.

Some of the commonly adopted architectures and models are discussed below.

#### **3.1.1 Loose Coupling VS Tight Coupling**

Traditional databases are monolithic (i.e. tightly coupled) in their design. Being monolithic means there is no separation of nodes or components of the database. For instance the file system, database engine, transaction manager, metadata and storage are all tightly coupled in a single node. Such a design decision makes it difficult to shard the system and can result in system downtime (unavailability) during an upgrade or maintenance [43]. NoSQL databases tend to separate system state from application state in order to provide high scalability [71]. System state includes metadata management which is crucial to the functioning of the system, while application state refers to the actual data of the application being managed [72]. For instance, the Google stack maintains a loosely coupled architecture consisting of GFS (File tier) [54], BigTable (record manager) [73], Megastore [51] (for transaction) and makes use of Chubby lock service [74] to manage the system state. Also, each cluster has a master server that manages placement of data on the other chunk (tablet) servers. Google also makes use of MapReduce [75] to process data on a large scale. PNUTS [60] also makes use of a loosely coupled architecture consisting of a tablet controller and message broker

to manage system state. The tablet controller manages location and relocation of tablets (shards) while message broker manages mapping of tablets with their replica. MongoDB consists of three server roles which include the router - called mongos which is used for routing requests to the cluster; the configuration server – saves cluster metadata; and the database server – called mongod which is used to store application data.

However, loose coupling also has its disadvantages. Communication and interaction among various components of the system can introduce network latencies, network partitions, and high traffic that may consume network bandwidth. However, this overhead is considered to be inconsequential when the performance gains of parallel processing are put into consideration.

### **3.1.2 Share Nothing VS Shared Disk**

Most cloud databases run shared-nothing architecture. In shared-nothing architecture, each node is responsible for a subset of the entire data and does not share any hardware component with other nodes. This enhances scalability and parallel processing. Shared-Nothing systems are known to scale faster and increase availability as there is no single point of failure [76]. Alternative, it is possible to have multiple processing nodes sharing a single storage disk which is known as shared disk. In spite of its advantages, shared-nothing architecture comes with numerous maintenance issues which include load balancing among nodes, complex 2-phase commit algorithm across nodes, and request routing amongst others [77].

### **3.1.3 Data Model**

As an implication of horizontal scale-out (sharding), NoSQL databases generally implement simple data models and can support only simple single-key operations [78]. There are no table joins or referential integrity between the entities stored in NoSQL databases. The choice of data model has a crucial implication in determining the type of queries (operations) that the application would be able to

perform. It is therefore important to consider the application domain which would best suit the database when designing a NoSQL data model.

### **3.1.4 Concurrency Control Techniques**

As stated earlier, concurrency control techniques help to maintain consistency and isolation in a database. NoSQL databases generally follow an eventual consistency model which means replicas could be out of date for a specified period of time. Resolving conflicts among replicas is therefore important for the system to achieve consistency – but eventually. The choice of where and how to resolve conflict is a critical issue. Key-value NoSQL databases tend to leave the task of conflict resolution to the application. This leaves a lot of burden to application designers. Dynamo [50] uses object versioning [79] to manage conflicts at application level. Spanner, a transactional database designed at Google, uses paxos protocol [15] to manage concurrency and prevent conflict, GFS uses namespace locking to manage concurrency. Some systems like ReTSO uses snapshot isolation (discussed later [section 5.1.1](#)) to avoid conflict.

### **3.1.5 Replication**

In cloud environment, there is a choice to be made on whether database systems should use synchronous (eager) or Asynchronous (lazy) replication mechanism [80]. Synchronous replication ensures that all replicas have the same view of data at all times. This ensures strong consistency across replicas (i.e. 1-copy serializability) but can affect availability if any one of the replicas is not available. Also replication can either be local or geographic. Cloud database vendors need to decide which of them will best suit their applications. PNUTS uses asynchronous geographic replication while BigTable supports eventually consistent replication across geographic clusters. There is also a decision to be made about which or how many servers can respond to ‘read’ and ‘write’ requests. Relational databases mainly make use of a primary-secondary replication algorithm in which case the primary server responds to read and write requests. The secondary replica only becomes active when the primary replica is down. In distributed

databases where replication factor is usually higher (say 3 or more), quorum approach is implemented. There are various mechanisms used to arrive at a quorum so as to guarantee consistency when responding to read requests [14] [81] [82].

### **3.1.6 Master-Slave VS Peer to Peer Architecture**

Generally, there are two main architectures used in cloud databases which include master-slave and peer-to-peer [83]. In a master/slave approach, the master manages the systems state, request routing and is aware of changes across all the nodes in the cluster. The master is also responsible for deciding the location of data and failure detection. In peer-to-peer architecture, all servers have the same roles and each server manages its meta-data and system state. Peer-to-peer servers are usually aware of information on other servers. Google systems make use of a Master-Slave approach with one server (Master) managing both metadata and data lease during updates on behalf of the system. Cassandra [57] and Dynamo [50] on the other hand, make use of peer-to-peer approach where all servers have equal role and duties. Dynamo controls traffic by storing routing information in each node. A request is sent to any node in the cluster and re-routed to the node that manages the data. Cassandra stores the node information also in Zookeeper [84] for recovery in the event of a failure. Master-Slave approach is prone to certain issues such as bottleneck issues when traffic is high [85] and outages in the event of a master failure. However, master/slave approach is able to easily manage consistency and it guarantees that requests are directed to the replica that has the most recent version. Systems that implement a peer-to-peer architecture on the other hand find it more challenging to guarantee consistency. The only way to guarantee that operations are directed towards the replica with the most recent version of data is by implementing synchronous replication which requires more effort [83].

### **3.1.7 Query Processing Approach**

In relational databases, the standard query language (SQL) is recognized as generalized language for querying all databases. In NoSQL databases, there is no

generally accepted query language. Also, NoSQL databases leave complex query processing to the application level and only offer simple operations. For instance, BigTable makes use of MapReduce [75] for processing jobs which is implemented using different programming language. Key-value databases tend to implement simple (put, get and delete) operation because of their data model. MongoDB [86] makes use of JSON statements for querying collections (Tables) via a JavaScript shell and also has driver support for querying the database using the API of other programming languages [37]. In NoSQL databases, data processing paradigms are categorized into three main groups namely batch processing, real-time processing and hybrid computation. The batch processing paradigm is appropriate when dealing with large volumes of data while the real-time processing is appropriate for dealing with data coming in at high speeds and in real time. The hybrid computation is a combination of real-time and batch processing loads.

Another important issue is how queries are routed to individual nodes. Since NoSQL databases make use of shared-nothing architecture and a single database can be scaled across thousands of nodes, there must be a proper and efficient mechanism for routing requests. In BigTable, queries are routed through the master server which stores information of node location. The master then directs the request to the specific node hosting the requested data item. In MongoDB cluster, requests are handled by the routing server which is also known as the mongos. The mongos receives and directs incoming requests to the appropriate shard or node. This is not usually the case with relational databases. Relational databases are monolithic and there are no specific roles among database nodes. Requests and metadata management are handled by the same node hosting the database.

### **3.1.8 Read Optimised VS Write Optimised**

The method of writing to disk affects performance and determines if the system is optimised for reads or writes. In BigTable, writes are appended to a single file (SSTables) per server and these files are immutable. This decision implies that writes in BigTable are optimized for writes as a write operation is appended to the end of a single file. A read operation, on the other hand, would have to scan through

the file to be able to locate the data. Indeed, the results in [73] shows that BigTable has a higher write throughput than reads. Dynamo [50] is optimized for reads because writes always involved disk seeks whereas a read request does not always involve disk access.

### **3.1.9 Latency VS Durability**

When data is persisted to disk, it achieves durability. However, the overhead associated with disk I/O incurs higher level of latency. If data is not persisted to disk first, the throughput increases but this can have an effect on the durability of transactions. However, some databases are designed to store their data permanently in memory in order to achieve high speed. Such databases are known as main memory databases (MMDB) [87]. In MMDBs, a copy of the database is usually stored in disk also. Most web applications make use of Memcached [88], an open source distributed memory caching system which reduces the load on a disk-based database by caching some of the information in memory.

The NoSQL database vendors generally implement a combination of the architectures discussed above, in their database designs. This combination determines the characteristics and properties of the database. It also determines the availability and consistency guarantees that the database can provide which in turn determines the suitability of a database for an application. The next section analyses a few of the popular NoSQL databases, their architectural implementation and the impact of these designs on their properties.

## **3.2 NOSQL DATABASES AND BIG DATA**

One of the primary objectives of NoSQL databases is to store and manage big data. But as data increases, it becomes more difficult for a single node (of a NoSQL database) to process big data [89]. To efficiently process data, NoSQL databases make use of the parallel processing paradigm (i.e. horizontal scaling).

To be able to scale data, the data is de-normalized and spread across multiple systems. In order to ensure that requests are always served (high availability), NoSQL databases implement weak consistency models. These decisions have the following implications on the operations of NoSQL databases [11].

- De-normalization of data implies that there is no referential integrity among data entities. Thus, simple data model is adopted.
- Unlike relational databases, there is lack of support for join operations in NoSQL databases. This prevents implementing complex queries. NoSQL databases generally offer only simple queries.
- Flexible schema adopted by NoSQL means rows can have different attributes i.e. with no strict table or database schema. This allows NoSQL to be ideal for supporting unstructured data but not structured data.
- The relaxed consistency models means there can be no support for transactions. NoSQL databases cannot support the ACID properties of transactions and are therefore inappropriate for applications that need strong consistency.

NoSQL databases are generally classified into four main types which are explained below [90].

**Document-Oriented databases** – Document databases store data in XML, JSON or BSON formats. Tables are referred to as collections and each row is called a document. Documents can contain multiple field attributes and each document can have different attributes implying that document databases have a flexible schema. Every document is indexed and has an associated key which is used to identify the document. Examples of document databases include MongoDB, CouchDB.

**Key-Value stores** – These are the most basic forms of NoSQL database. Each record is stored as a key-value pair where the value could be a BLOB object that the database stores without necessarily knowing what type of data or what is inside the value. Example of Key-value stores includes Dynamo, Riak, Redis. Key value stores support only basic operations such as get and put.

**Column stores** - Column databases store data in columns and each row can have different number of columns. Furthermore, column stores introduce what is known as column families where data that is associated together can be grouped together to form a column family. Column databases can be used to store structured and semi-structured data. Examples of column databases include BigTable, HBase and Cassandra.

**Graph databases** – These databases are used to represent objects and relationships that exist between the objects. A node represents an object and the edges represent the relationship between objects. In graph databases, the relationships represent an important aspect of the database and provide the value that can be derived from the database.

Each of these classes of NoSQL database are more ideal for some applications than others [91]. The next section examines some of the common NoSQL databases and their characteristics.

### 3.2.1 BIGTABLE

BigTable is a NoSQL database designed by Google that stores data for applications such as Google Earth and Google Finance. It belongs to the column-oriented class of NoSQL databases. In BigTable, data is ordered lexicographically by row key and partitioned into different nodes on the row keys using range-partitions. This makes it suitable for applications that make use of sequential reads and writes. Data is indexed by row key, column key and timestamp and tables are stored in SSTable file format in GFS. The application state is managed by the Google file system (GFS). GFS divides its files into large chunks of 64MB and uses Master-Slave architecture. Files are also replicated across multiple slave servers. The master monitors the activities of other servers, detects failures among nodes (e.g. using regular *HeartBeat* messages) [54] and manages meta-data. But it is not directly involved in reads and writes. During failures, the process of re-replication is prioritized based on factors, such as, how many replicas of the data are alive. GFS uses a relaxed consistency model that supports most of its applications. BigTable also uses chubby service [74] to manage its system state. BigTable allows users to group sets of columns frequently accessed together into locality groups



to speed up data scans. When a client makes a request, the request is forwarded to the master. The master is aware of the location of all data items and it responds to the client with the location information of the needed data item. The clients then push all the updates to the required node and replicas and proceeds to write the data.

A perceived weakness in BigTable is that it is not very efficient for applications that have complex and evolving schemas. It is also limited in providing wide area replication [32, 49]. Due to these limitations, Spanner was designed. Also, BigTable implements an eventually consistent model across replicas. Hence it is not ideal for applications that need strong consistency requirements.

### **3.2.2 MONGODB**

MongoDB belongs to the document class of NoSQL databases. It stores data as documents in binary representation called BSON. Documents are organized into table structure, which is referred to as a collection. MongoDB like most other NoSQL databases has flexible schema model. MongoDB supports three types of partitioning (or sharding) namely: Range-based, Hash-based and Tag-aware sharding. In tag-aware sharding, the user specifies a configuration for grouping key ranges together. MongoDB automatically balances load in the cluster. Each data item (document) has an ID which can be indexed to enable faster queries. A MongoDB cluster consists of three server roles which are the router (Mongos), configuration server and the database shards (mongod) or replica set. A database shard stores a subset of the data, the config server stores metadata and information on data locations while the mongos server acts as a router and routes requests (read and write) from the application to the shards. During operations, queries are sent to the router server (mongos). The mongos server then directs the query or update to the shard that stores the data. The mongos gets information of data location from the configuration server and caches it. The mongos itself has no persistent state. MongoDB uses write-ahead logs called journal to ensure durability and recovery. The number of replicas is configurable and mongod sets one of the replicas to be the primary while the others are secondary.

### 3.2.3 DYNAMO

Dynamo [50] is a key-value store designed as a storage solution for the Amazon Simple Storage Service (Amazon S3). It provides a BASE level consistency model but no isolation guarantees. Dynamo uses a peer-to-peer architecture where all nodes have equal responsibility and as a result, there is no single point of failure. Each node has information about some other node in its range. Dynamo uses consistent hashing as partitioning algorithm [92] and depends on the client for reconciling different versions of an object. It uses 3 replicas by default and uses a gossip-based membership protocol [93] for failure detection. In gossip-based protocol, nodes exchange information with each other allowing for failure detection when there is no response from any node. Operations in Dynamo are limited to single key operations which include get (key) and put (key, context, object) where context represents metadata and Dynamo does not support transactions. This is in sharp contrast with BigTable [73] which stores system metadata at the Master. Dynamo is used to manage Amazons shopping cart application and it ensures that customer never lose any item they place in a shopping cart. To achieve this, each key item has a preference list of top N nodes that host replicas of that key. When there is a node failure, read and write operations would still continue on any of the nodes in the preferences list for that key item. Dynamo then makes use of a form of object versioning technique that merges data in divergent replicas (which could be as a result of failure) to ensure that no item is lost. Dynamo uses the formula  $R + W > N$  to maintain consistency among replicas ( $N$  = minimum number of nodes that stores the object,  $R$  = minimum number of nodes involved in a read and  $W$  = minimum number of nodes involved in a write).

Dynamo has some perceived weaknesses. It was designed to be an in-house database to be used only in trusted environment and hence has limited security mechanism. Another weakness of Dynamo is that it relies on application logic to resolve conflicts as it can only provide eventual consistency. The designers have chosen to view this as strength because it allows application designers the flexibility to determine the logic that works for their applications. But in reality, it is a weakness as application designers have extra responsibility of programming

consistency logic. Also, operations in Dynamo are optimized to handle small data objects typically less than 1 MB.

### **3.2.4 CASSANDRA**

Cassandra is another storage solution designed by Facebook to meet reliability and scalability needs as well as to handle very high write throughputs which is typical of Facebook application. It belongs to the column class category of NoSQL databases. It also runs on cheap commodity hardware. Cassandra uses a form of weak consistency model. Its architectural feature is a mixture of features from BigTable and Dynamo. Its data model is very similar to BigTable but it uses peer to peer architecture like Dynamo. It also makes use of consistent hashing as its partitioning algorithm (as in Dynamo). Cassandra introduces certain features like super-column and column family and data is accessed using the arrangement column-family: super-column: column. Cassandra uses a system called zookeeper that stores data placement information and metadata in a fault tolerant manner. This will help a recovering node know the ranges of data it is responsible for.

### **3.2.5 PNUTS**

PNUTS [60] is a distributed database system used for Yahoo!'s web application. PNUTS uses a data model similar to relational databases. The design requirements for this system was to achieve high scalability (a key requirement for web applications), low latency in accordance with Yahoo!'s SLA, high availability and fault tolerance. Therefore, to achieve high scalability, there is no referential integrity enforcement across tables. Most of Yahoo!'s applications manipulate one record at a time and supports a relaxed consistency model. In PNUTS, data is asynchronously replicated over geographic locations. Therefore on the CAP spectrum, PNUTS is Partition tolerant and high availability sacrificing consistency. The implementation of PNUTs requires two additional machines to serve as configuration server and router. The router server directs request to the particular server that hosts the data. The configuration server also called tablet controller stores and manages the mapping of data to the respective servers. PNUTS uses

the Yahoo Message Broker (YMB) to manage consistency. When a data item is published to the YMB, it is considered as committed. Updates are then asynchronously propagated to other replicas. To manage inconsistencies across replicas, PNUTS uses timeline consistency earlier discussed in [section 2.7.2](#). To achieve this, each data item has a nominated master replica. Only a master can accept updates request from clients. Each version of a data item also has an increasing sequence number to identify the outdated data.

As mentioned earlier, NoSQL databases do not offer support for transactions. Also, a review of the existing databases show that each of these systems show that they can only support simple operations [94]. However, in recent times, efforts have been made to implement transaction in NoSQL databases. The next section examines the various ways in which this has been achieved.

### **3.3 TRANSACTIONS IN CLOUD DATABASES**

In cloud environments, partitioning a database into shards improves performance and availability. But it also makes transaction processing more complex, particularly transactions involving multiple data items. This is because, a single database is split up across multiple nodes and each node is responsible for processing its own data (shared-nothing) without a centralised coordinator that can coordinate between different nodes of a NoSQL database. This is contrary to the classical distributed database wherein operations can take place between two or more different database management systems and are usually coordinated by a single database (known as the coordinator) using protocols such as two phase commit protocol. In cloud, coordinating operations in a single database that scales across multiple systems (or nodes) is a challenging task [95].

Various approaches have been developed in order to solve the problem of transactions involving multiple data items in NoSQL databases. These approaches can be divided into three main categories [96] which include: (i) integrated approach, (ii) middleware approach and (iii) API approach. These approaches are discussed and analysed in the following sections.

### **3.3.1 The Integrated Approach**

This approach involves building transaction support into the cloud data store. In other words, NoSQL databases should be designed such that there is a support for transactions. Examples of cloud databases that use this approach include Spanner [97] and COPS [98]. These are discussed below.

#### **3.3.1.1 Google Spanner**

Spanner [97] is a globally distributed database built by Google to support consistent transactions across a globally distributed environment. Spanner was designed to improve on the shortcomings of BigTable [73] and to be able to manage applications that have complex structure and need strong consistency. Spanner is accessed through an API that implements read-only, read-write and snapshot reads transactions. It provides support for externally consistent reads and writes and globally consistent reads at a specific timestamp. Externally consistent transactions guarantee that they will always receive current information [99]. Every deployment of Spanner is implemented in an abstraction called the Universe. A universe is divided into zones. Each universe consists of a universe master and a placement driver while the zones consist of one zonemaster, one location proxy and up to thousands of spanservers. The universe master provides status on zones and the placement driver oversees movement of data across zones. A timestamp is ascribed to data on commit (meaning that there can be multiple versions of a data item) and every Spanserver in each replica maintains a lock table for concurrency control. Spanner implements a timing mechanism API called TrueTime which uses GPS and atomic clocks to measure timing. Each datacentre contains one time master and each machine in a datacentre has a timeslave daemon. Time master machines regularly compare their times against each other to ensure synchronization between them. The timeslave on each machine would check its time against a number of masters and any machine whose local clock is larger than a given threshold is evicted. With this

timing in place, Spanner can control transactions between spanservers using the start timestamp and commit timestamp of transactions. Each set of replica is referred to as a paxos group and each group has a leader. During transactions, a leader among participant leaders replica is chosen as the coordinator of that transaction. Spanner introduces the concept of hierarchy at the level of the table to enforce relationships. The top level table is referred to as a directory table and each row key in a descendant table starts with the key of the directory table.

### **3.3.1.2 Cluster of Order Preserving Servers (COPS)**

COPS [98], is a database that provides causal+ consistency over a wide-area distribution. Causal+ is defined as a combination of causal consistency and a convergent conflict handling mechanism. Causal consistency ensures that the causal dependencies between the data (keys) in a database are preserved. The conflict handling mechanisms guarantees that replicas never remain permanently divergent by applying, update operations in the same order across all replicas. In order to achieve this, COPS introduces two variables known as 'versions' and 'dependency'. Each data key can have multiple versions and is denoted as  $Key_{version}$ . Updates to replicas always produce an increasing (or later) version of a key to preserve causal consistency. This is referred to as progressing property in COPS. The dependency variable on the other hand refers to the ordering. For instance, if in a data store,  $X_i$  precedes  $Y_j$ , then  $Y_j$  depends on  $X_i$ . COPS can then enforce causal consistency by ensuring that updates are written only after all its dependencies/ dependent keys have been written. Therefore, the key version is used to enforce ordering among different versions of a data (or key) item, while the dependency is used to enforce order across different keys. Convergent conflict handling ensures that conflicting operations are handled in the same manner across all replicas ensuring that the outcome of the conflict must be the same across all replicas. This is achieved by using well know techniques such as last-writer-wins rule.

Each cluster is operated as a strongly consistent key-value store with key spaces partitioned among nodes. A datacentre contains two replicas of the cluster. One of the replicas is known as the local (primary) cluster of a datacentre and the

other is a secondary replica. The local cluster is a strongly consistent with its replica (synchronous replication) in the datacentre while replication between clusters in different datacentres is asynchronous. The term 'Equivalent nodes', is used to refer to the set of primary nodes across all clusters. When a write (or update) operation is performed on a local primary node, the updates are sent asynchronously in a queue to all equivalent nodes in other data centres. The equivalent nodes then enforce causal+ consistency in their update operations based on the information provided from the dependency list. COPS is implemented as a loosely coupled architecture which consist of two components. They include: (i) Key-value database and (ii) a client library that exports 'get' and 'put' operations. The key-value database also stores metadata such as the key version number and implements slightly more complex operations such as 'get\_by\_version', 'put\_after' and 'dep\_check'. These features enable COPS to maintain causal+ consistency despite its asynchronous model of replication. COPS-GT, is a flavour of COPS with an extra operation referred to as 'get\_transaction'. In COPS-GT, each key is mapped to a version and a dependency value in the form:  $\text{key} \rightarrow \langle \text{version}, \text{value}, \text{dependency} \rangle$ . The dependency, which is of the form  $\langle \text{key}, \text{version} \rangle$  tells the node which key that a data item depends on. This helps it to implement ordering across keys. The client library of COPS-GT keeps information about dependencies in a 'context' parameter which is associated with every operation and is identified by a 'context\_id' attribute. The context parameter is stored in a table and used to track dependency across operations. A 'get' request will normally include a key and a dependency while a put request stores the version number to the 'context' parameter.

COPS, like most systems, has its short comings. The process of enforcing causal ordering over a wide area network is non-trivial and bandwidth intensive. This will make it impractical when the number of data centres involved is high. Also, failures in a datacentre could mean that updates not yet propagated to remote datacentres could be permanently lost since COPS uses asynchronous replication across datacentres.

### 3.3.2 The Middleware Approach

A second approach is to execute transactions on cloud database using a middleware. Megastore [51], G-Store [100], CloudTPS [101] and Deuteronomy [102] are examples of this approach. The implementations of Megastore and G-store are explained below.

#### 3.3.2.1 Megastore

Megastore [51] was built by Google with the objectives of achieving the scalability capabilities of NoSQL databases and the consistency guarantees of traditional database. The requirements that led to the design of Megastore include high scalability, consistent view of data, low latency and high availability. Megastore uses Paxos [15] algorithm to provide fault tolerance among replicas. In Megastore, data is partitioned into 'entity groups' and each entity group contain a set of keys which are synchronously replicated over wide geographic area. Within an entity group, ACID properties are enforced. Operations across groups are asynchronous and are sent in a queue. For example, an email account forms an entity group in Megastore. Thus, operations within an email account would be ACID level transaction but operations across email accounts make use of asynchronous messaging. Megastore uses BigTable as its back-end NoSQL database. To enforce relationships amongst tables in an entity group, Megastore makes use of child-root table schema. Therefore, each child table must have a key that references its root table. Megastore tends to cluster keys that are read together and maps each entity to a single row arranged in contiguous order in BigTable. The major difference in the data models of Megastore and traditional relational database is in the way keys are physically stored. Since BigTable does not support table joins, the key of each row is derived by concatenating the keys of the child and parent tables in a row. Megastore exposes two types of indexes namely; Local index, used within an entity group and Global index, used to search for entities when the entity group is not known in advance. Reads and writes can be processed from any replica; and as such there is no notion of a fixed primary replica. This allows for higher availability, faster read and write operations thereby



reducing latency since applications can easily access the replica closer to them. This is achievable because Megastore makes use of Paxos to manage updates to replicas. The accessed replica represents the leader for that transaction and the logs are then replicated synchronously to a quorum of replica.

As noted earlier, ACID transactions are only achievable within an entity group. This is a known limitation of Megastore. Also, Megastore entity group membership is static in nature and as such, keys that belong to an entity group must remain a member of that entity throughout their life time. This means that there can be no ACID transactions between keys that belong to different entity group. This makes it impractical for certain applications that need ACID operations across different keys that don't belong to the same entity group. It is also unsuitable for applications that need dynamic grouping over the period of their life time. An example of such application includes online game applications that need grouping of keys for the duration of the game alone. G-store attempts to address this limitation. Megastore is also known to have relatively poor write throughput because of its synchronous replication within entity groups [97].

### **3.3.2.2 G-Store**

The design consideration for G-Store includes high scalability, high availability and fault tolerance as well as multi-key transactional access. G-Store uses the same concept as Megastore which uses a Key Grouping protocol [100] to group keys for applications that need multi-key transactional access. One feature however of G-Store is that grouping of keys is dynamic and a key can belong to different groups during its life time but only one group at any given time. Groups are formed by the applications. Keys of the same group are transferred into a single node for the period of their membership. This is to prevent the complexity involved in distributed synchronization. In G-Store, there is a concept of leader and follower keys. Every group formed has a leader while other keys of the group are known as follower keys. There are two phases involved namely: Group creation phase and Group deletion phase. The group creation phase is initiated when an application client chooses a leader. The leader in turn sends a join request to all members of the group after it has logged the list of members. The node where the leader is

located is known as the owner of the group. Like the 2PC, the group creation phase occurs in two phases. The basic protocol is highlighted below.

- Leader sends a Join Request {J} to the followers – To acquire ownership
- Followers respond with a Join Ack {JA} message

When all members have joined, the group creation phase terminates. After a group has been created, transactions can take place during the life time of the group. G-store can only provide ACID transactions within the group and transactions need not span more than one node. During the group deletion phase, ownership of individual keys is transferred from the leader back to each of the followers. The client sends a group delete request to the leader, the leader then logs the request and sends a delete request to all the followers. In the basic protocol, the followers do not need to respond to a delete request. The protocol was further optimized to deal with failure, concurrent group creation and recovery. In the optimised protocol, every group has a group id and a yield id for every operation and G-Store logs this information using write-ahead logs which is useful for recovery.

G-store can be implemented as a client based implementation and also as a middleware to a Key-Value store. One limitation of G-store implementation of transactions is that there is a high level of overhead during the group formation stages.

### **3.3.3 The API Approach**

A third approach is to provide transactional access to the data i.e. client applications access the data through an API that implements transaction semantics. Examples of this implementation include Percolator [103] and ReTSO [104].

#### **3.3.3.1 Percolator**

Percolator was built at Google partly to address the short-comings of BigTable [18] which lacks support for multi-key transactions [32]. Percolator is designed to

process incremental update for web indexing. Percolator provides ACID transaction support with incremental computation. Percolator itself is made up of three components namely: (i) Percolator worker, (ii) BigTable server and (iii) GFS chunk-server. Every node in a Percolator cluster contains these three components. Percolator uses snapshot isolation to implement multi-key transactions and stores multiple version of each key using Bigtable's timestamp to identify versions. Transactions make use of a distributed lock system and a timestamp oracle (TO). The timestamp oracle is a server that issues an always increasing timestamp which guarantees that transactions are properly ordered. To perform a write operation, a lock is requested on all the rows involved in the write. The client then uses a timestamp oracle to retrieve its commit time after which it releases its lock. A transaction will abort if it can see a lock or a write that has occurred after its own start timestamp. Therefore, every transaction must contact the timestamp oracle twice and the highest allocated timestamp is kept in stable storage. This increasing timestamp will guarantee that a 'get' request will return only writes that committed before the transaction start timestamp.

Percolator lacks a global deadlock detector. This can cause an increase in latency when there are conflicting transactions. Therefore, Percolator is not ideal for environments that need extremely low latency.

### **3.3.3.2 ReTSO**

ReTSO [104] is used to support client side transactions for large scale storage systems. ReTSO makes use of a centralized Transaction Status Oracle (TSO) to implement Snapshot Isolation. In Snapshot Isolation, transactions are carried out on a snapshot of the data as at the time of the transaction. ReTSO uses a system called BookKeeper [105] to persist write-ahead logs in order to achieve higher availability. Before a transaction starts, it must receive a start timestamp request from the TSO. The TSO manages incoming transaction request and checks for conflicts between transactions. ReTSO generates a start timestamp and commit timestamp for all transactions on all servers using a Timestamp Oracle to guarantee integrity of the timestamp (i.e. timestamps ordering must be unique and incremental). The TSO stores the status of all active transactions and can be

queried to verify the status of any transaction. A write request generates a start timestamp on the TSO (the timestamp represents the version number of that data) and is saved in a 'PendingWrite' Column in memory. Once the data is committed on the Key value store, the TSO generates a commit timestamp which is sent to the client. The client can then clean up its 'PendingWrite' Column after updating its commit timestamp. In the case of an abort, the client deletes the data and also cleans up the 'PendingWrite' column. A read operation observes the last committed data before its own start timestamp.

Using a centralized Transaction Status Oracle can be a bottleneck in distributed systems. More importantly, when a transaction is trying to retrieve a commit time, the TSO may need to check its memory to be sure that there are no conflicting transactions. This can lead to long and unnecessary waits. ReTSO addresses this by limiting the amount of information kept in memory. ReTSO also uses replicated write-ahead log across multiple dedicated storage devices (BookKeeper) to prevent loss of data. It should be noted that the BookKeeper is dedicated to this task alone to achieve high performance.

### **3.4 ANALYSIS OF OTHER TRANSACTION MODELS AND PROTOCOLS**

In addition to the above, various other approaches have been developed in order to implement transactions in high scalable cloud databases. This section reviews and analyses some of the common approaches. In [106], transactions are implemented by adapting a relational database into a shared-nothing architecture used in NoSQL systems. In their approach, they limit transactions to execute on a single node thereby avoiding the need for two-phase commit protocol. A database contains a group of tables called a table group. Each node contains a subset of the entire database which is a group of tables that are joined by a column. The column is known as the partitioning key. Each table group must be able to fit into a single node, such that the system can only provide ACID transactions support for data within a single node. Data is replicated in the cluster but one replica serves as the primary replica. The primary replica is also used to handle updates.

A Deuteronomy approach is proposed in [102]. The architecture used in [102] is similar to the architecture proposed in this thesis, however Deuteronomy makes use of data locks which incurs considerable overhead [107]. Deuteronomy system contains two components which include the transactional component (TC) and the data component (DC). The TC manages transactions and concurrency control while data is stored on the DC. Transactions in Deuteronomy can span multiple DCs. Applications send their requests to the TC. Since the TC stores table information (meta-data) and manages session, the TC knows which DC is hosting the requested data. When requests are submitted to the TC, the TC sends the operations to the required DCs. The DCs perform the necessary operations and the TC logs the operations after they have been concluded.

Elastras [108] [109] consist of three components which include the Transaction Manager (TM), Metadata manager and the distributed storage layer which does not support transactions. The storage layer manages issues such as replication and fault-tolerance and implements an eventually consistent model of replication. The transaction manager is further divided into two layers namely the Higher Level Transaction Manager (HTM) and the Owning Transaction Manager (OTM). The OTM has exclusive access rights to a subset of data in the distributed storage layer and caches some of the data. Requests are sent to a HTM, the HTM then routes the request to the appropriate OTM in charge of the requested data. If the data is in the OTMs cache, the OTM performs the updates, otherwise, it requests for the required data. The OTM also uses write-ahead logging to perform recovery. However, Elastras can support only mini transactions defined in [110]. A mini transaction allows users to atomically batch together updates and to conditionally modify data in multiple nodes. For instance, with mini transactions, a transaction can be regarded as committed transaction on the condition all the operations of that transaction will be successful. If any of the operations are aborted, the transaction can then be rolled back. This is called a conditional commit and can only work for certain types of applications.

Warp [111] provides a one-copy serializable transaction support implemented via a client library that supports linearizable transactions, i.e., transactions are executed in the order they arrive. The system consists of a client library, storage servers and a coordinator server which maps key ranges to storage servers. It

implements a protocol that identifies conflicts in transactions by maintaining a dependency graph across transactions.

All of the systems described above have some limitations in their ability to perform transactions. Most of the systems are unable to provide ACID transactions. They only provide some form of limited properties of transactions which makes them unsuitable for applications that need ACID.

### **3.5 DISCUSSION AND CONCLUSION**

This chapter investigated into the existing literature in order to provide an insight into the different aspects of data processing in cloud environment which include: architectural consideration and data models of cloud or NoSQL databases, the different types and characteristics of NoSQL databases, and the transaction management techniques developed for NoSQL databases. It is observed that multiple factors (such as architecture, data models, classes of databases, transactions models, etc) have impact on the performance, reliability, availability, and consistency of NoSQL databases. It is also examined that each of the classes of NoSQL database are more ideal for some applications than others. For instance, BigTable, a column-oriented database, is used in Google Earth and Google Finance. Similarly, MongoDB, is used in retail and online news applications such as Ebay and Forbes online magazine.

This chapter then critically reviewed existing transaction management techniques which have been implemented in various industry (commercial) NoSQL databases as well as prototypical or research-based NoSQL databases. The review of the existing techniques showed that there is not a single transaction management technique that provides all the features such as improved performance, availability, consistency and so on.

All of the approaches reviewed above have the main shortcoming (or trade-off) that these systems were designed with particular applications in mind. According to [94], the NoSQL databases will not replace the relational databases but will be better fit for certain applications. The NoSQL databases generally offer only

simple operations and will definitely work with certain applications. Dynamo [50] for instance offers no data integrity guarantee and employs a very weak model of consistency. BigTable [73] has poor write performances and very limited query capabilities (as with dynamo). Cassandra [57] is inefficient for processing ad hoc queries. A review of NoSQL systems to offer transaction support shows that most of the existing systems have limitations which will make them unsuitable for certain applications [112]. These trades-offs however have an impact on the performance of the systems. In [113], the system makes use of data locks which is a pessimistic concurrency control mechanisms that reduces throughput and involves a high level of overhead. Megastore [51] requires data to be partitioned into groups and can only provide ACID transactions within groups. CloudTPS [101] makes use of two phase commit which could be cumbersome. G-Store [100] improves on megastore by making groups more dynamic such that any group can contain different data at different times reducing the need for two phase commit protocols. However, transactions are still limited to within entity groups. All these systems have certain performance issues that our proposed system intends to improve on.

As discussed above there exist a number of research challenges in cloud and NoSQL databases. But the work in this thesis focuses on the following main research issues.

- It has been observed that current research approaches and commercial NoSQL databases do not enforce strict consistency in big data processing and management. This is a consequence of providing high performance and high availability of big data in the current solutions.
- It has been identified that current designs of NoSQL databases do not support normalization and integrity constraints. This leads to the facts the complex queries and transactions are not supported by the current NoSQL databases.
- It has also been observed that current solutions scarify the support of transactions and the implementation of ACID properties in NoSQL databases for achieving scalability and efficiency.

The remaining chapters describe the proposed approach that aims to address the above research issues.



# CHAPTER 4

## **MODELLING AND DESIGN OF THE PROPOSED APPROACH – NoSQL-TX**

This chapter explains the theoretical model and design of the proposed transaction approach which is referred to as NoSQL-TX. The acronym NoSQL-TX is inspired from WS-TX which is used to describe web services transactions [114]. Section 4.1 defines and specifies the constraints of the proposed model. Section 4.2 explains snapshot isolation which is the technique implemented by the proposed system. Section 4.3 describes the architecture of the system as well as the various components that make up the system. The approach of NoSQL-TX follows the middle ware approach explained in [section 3.3.2](#). The transaction state diagram which models a transaction life-cycle and the protocols for a commit operation in the multi-key transaction model is explained in section 4.4.

Section 4.5 explains the interactions that take place among the components of the system to execute a transaction. Section 4.6 explains the protocol followed by the system to perform a transaction commit and section 4.7 explains the various scenarios that can cause a transaction to abort. Finally, section 4.8 explains the protocol for replica management implemented by the prototype system.

### **4.1 NoSQL TRANSACTIONS**

The ACID properties of transactions are the standard and most commonly used properties of database transactions. One of the objectives of the proposed model is to implement ACID properties in NoSQL databases. Before specifying the constraints and definitions of our NoSQL transaction model, a brief summary of ACID properties is re-emphasized.

**Atomicity:** Implies that all the operations in a transaction must be successfully executed or none of them must execute at all. In other words, a transaction is considered as an atomic unit of operations and the failure of any operation in a transaction means that all successful operations must be rolled back.

**Consistency:** The database must remain in a consistent state after the execution of a transaction. Therefore any updates made to data by a transaction must transform the database from one consistent state to another consistent state.

**Isolation:** Transactions must not expose their intermediate results to other concurrently running transactions. This means that the activities of a transaction must not affect the result of other on-going transactions.

**Durability:** Results of a completed transaction must be made permanent in the database store in order to provide fault tolerance in the event of failures.

Fundamental definitions of the proposed transaction model are illustrated as follow.

*Definition 4.1:* A NoSQL transaction *NST* is defined as the execution of a (cloud) application which comprises different operations that provide transitions between (partially) consistent states of the shared data. Therefore, *NST* is a sequence of operations which are executed in a way such that all of them are successfully completed or none at all.

*Definition 4.2:* A *NST* is a multi-key transaction as it involves more than one data key item and one or more operations.

Recall that NoSQL databases do not perform multi-key operations. Rather, they support only simple single key operations such as a *get()* or *put()*.

Based on the above, *NST* is formally defined as a tuple,  $NST = (OP, PaO)$ , where *OP* is a set of operations,  $OP = \{OP_i \mid i = 1 \dots n\}$ , and *PaO* is a partial ordering of the operations which determines their order of execution. For instance,  $OP_i > OP_j$  represents that  $OP_i$  is executed before  $OP_j$ . The partial ordering comes from the

fact that transactions do not necessarily commit in the order in which they arrive. Order is only enforced when there is a conflict between two transactions.

In the proposed model,  $OP_1^r[DE]$  represents a read operation of NST; meaning that NST reads data entity,  $DE$ , from a NoSQL database. Similarly,  $OP_1^w[DE]$  represents a write operation of NST; meaning that NST writes (updates) a data entity,  $DE$  to a NoSQL database. The read and write operations above ( $OP_1^r[DE]$  and  $OP_1^w[DE]$ ) are used to model the CRUD (Create, Read, Update and Delete) operations which are most commonly implemented in NoSQL systems. Note that NoSQL databases adopt CRUD operations from traditional databases.

In the proposed model  $OP_1^r[DE]$  represents the Read (of CRUD) and  $OP_1^w[DE]$  represents the Create, Update and Delete operations (of CRUD).  $OP_1^r[DE]$  is simply to read data without any modification to the data.  $OP_1^w[DE]$  is to write data meaning that data can be modified through Create, Update or Delete operation. In addition, to data read/write operations, NST is also associated with (control) operations, begin or start, commit and abort. These are explained as follow.

*Begin or start operation:* The execution of each NST must be marked through a begin or start operation. That is, NST should begin first before any of its operations ( $OP = \{OP_i \mid i = 1\dots n\} \in NST$ ) can be executed.

*Commit and Abort operations:* Each NST terminates with either a commit or an abort operation. If NST is successfully executed then it terminates with a *commit* operation. If NST cannot be successfully executed then it terminates with an abort operation.

NST can be of type seq (*Begin* |  $OP_i$  | *Cmt* | *Abt*) but with the condition that either *Cmt* (commit) or *Abt* (abort) occurs only once within the sequence. The events/scenarios that can lead to the system aborting a transaction are explained in later sections. Based on the above, the constraints on NST and its (begin, read, write, commit and abort) operations are specified below.

A NST comprises of different read/write operations but can have either one commit or abort operation. This is denoted as:

- $NST_i = \{Begin\} \cup \{OP_1^r[DE], \dots, OP_n^r[DE]\} \cup \{OP_1^w[DE], \dots, OP_n^w[DE]\} \cup \{Cmt_i, Abt_i\}$

Where  $DE$  is Data Entity.

- $Cmt_i \in NST_i$  iff  $Abt_i \notin NST_i$  i.e., If  $NST_i$  is committed then abort operation cannot be executed.
- Assume a (control) operation,  $ct_i$ , is  $Cmt_i$  or  $Abt_i$  (transaction commits or aborts), then for any read/write operation  $OP_i[DE] \in NST_i$ ,  $OP_i[DE] > ct_i$ . In other words, commit or abort operation must be after the read/write operation.
- If  $OP_1^r[DE], OP_1^w[DE] \in NST_i$ , then such read/write operations should be ordered either as  $OP_1^r[DE] < OP_1^w[DE]$  or  $OP_1^w[DE] < OP_1^r[DE]$ . That is, data entity,  $DE$ , should be read and written in a proper order.
- If Transaction **A** contains a data entity  $[DE_\alpha]$  and Transaction **B** also contains the same data entity  $[DE_\alpha]$ , then both transactions cannot commit at the same time. In other words, the commit time,  $T$ , of Transaction **A** cannot be equal to the commit time of Transaction **B**, i.e.,  $T_{A\ commit} \neq T_{B\ commit}$

## 4.2 SYSTEM DESIGN APPROACH

The proposed design aims to achieve high availability, and efficiency (or performance) but without sacrificing consistency of data. To achieve high efficiency, the proposed design takes advantage of proven database concurrency control techniques such as Snapshot Isolation [115]. But this is enhanced with the strength of scalable shared-nothing architecture of NoSQL databases. Before proceeding to explain the architectural decisions, the technique of snapshot isolation is explained in details below.

### 4.2.1 Snapshot Isolation

As stated earlier, snapshot isolation is an optimistic concurrency control mechanism that ensures that transactions are never blocked. In snapshot isolation, transactions perform read and write operations on a snapshot of the data. This means that a read operation can only read data that has been

committed at the time of the read. A write operation can only succeed if no other write operation has committed on the same data during its lifetime. The lifetime of a transaction is the period between the transactions start time and commit time. Each transaction has a start timestamp and a commit timestamp. A transaction  $T_i$  starts by obtaining a start timestamp  $T_{i \text{ start-time}}$ . Before  $T_i$  commits, it tries to obtain a commit timestamp  $T_{i \text{ commit-time}}$ . In this scenario, the transaction  $T_i$  would be successfully committed only if there has not been any other transaction  $T_j$  whose commit timestamp falls within the period/interval of start timestamp and commit timestamp (transaction lifetime) of  $T_i$ . If the following situation happens:

$T_{i \text{ start-time}} \rightarrow T_{j \text{ commit-time}} \rightarrow T_{i \text{ commit-time}}$ , and both transactions write to the same data, then  $T_i$  must abort (Where  $\rightarrow$  represents control /order of flow).

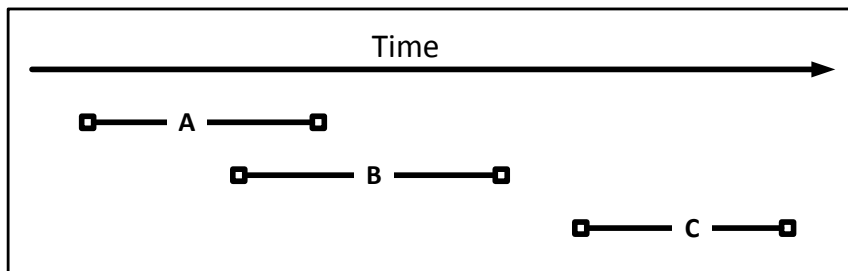


Figure 4.1: Snapshot Isolation

To explain further, assume that three transactions A and B and C (see Figure 4.1 above) are trying to modify a data entity  $[DE_\alpha]$ . Transaction A would commit even though it conflicts with B. This is because snapshot isolation follows the “First-Committer-Wins” rule. Transaction B reads data entity  $[DE_\alpha]$  which may (or may not) reflect the latest update written by transaction A. Transaction B tries to commit a modification on the data entity  $[DE_\alpha]$  but A has updated the data item data entity  $[DE_\alpha]$  and committed before transaction B commits, then B must abort (notwithstanding the start time of A). This way, consistency among different transactions is preserved. Note that transaction B will only abort if it is in conflict with transaction A (see definition of conflict in [section 2.2](#)). The commit time for every data item is also stored alongside with the data item as a parameter. This

will enable the system to maintain consistency in the presence of concurrently running transactions.

Systems implementing snapshot isolation in cloud environment would normally have a form of time ordering among nodes to be able to determine the start time and commit time of transactions. Spanner [97], ReTSO [104] and Walter [116] all make use of various forms of snapshot isolation.

#### 4.2.2 Rationale for Snapshot Isolation

The decision to implement snapshot isolation as a concurrency control mechanism is due to the fact that it provides a much higher concurrency than using classical locking systems. Snapshot Isolation never delays or blocks any read. Also, locking imposes high processing overheads [41] [117] on databases since any write operation must obtain a lock first even if there are no conflicting transactions.

Also, snapshot isolation avoids the following three anomalies [118].

**Dirty reads** - This occurs when a transaction **A** updates a data item and another transaction **B** reads the data item before **A** commits or rollbacks. If **A** rollbacks, transaction **B** would have read a wrong data since transaction **A** did not commit.

**Non-repeatable reads** - As the name implies, non-repeatable reads means that a repeated read by a single transaction returns different value. Assume a transaction **A** reads a data item and another transaction **B** either modifies or deletes that data item. If transaction **A** performs another read on the same item, it will yield a different result from the initial read of transaction **A**. This means that a transaction that contains multiple read operations can return different results for a given item.

**Phantom reads** - Assume that a transaction **A** reads a set of data which satisfies a predicate condition supplied by a user. Another transaction **B** then inserts new data items that also matches the predicate conditions stated in transaction **A** and is committed. This introduces new sets of data that also satisfy the conditions stated in transaction **A**. Repeating read in transaction **A** would produce a set of items that differ from the initial read of **A**. This is called a Phantom reads.

### 4.3 ARCHITECTURE OF THE NoSQL-TX

The architecture of the proposed system – NoSQL-TX, follows a loosely coupled architecture described in [section 3.1.1](#). The loosely coupled architecture separates the mechanisms of transaction processing from data storage (see Figure 4.3). As mentioned earlier, the system comprises of three main components which include the Data Management Store (DMS), Transaction Processing Engine (TPE) and the Time Stamp Manager (TSM). The functions of each of these components are explained in next section. The architecture allows the system to be scalable in two dimensions. First, as the size of data increases, the number of nodes at the data management store can be increased. Second, as the number of transactions increase, the number of transaction processing engines can also be increased in order to meet up with the demand.

#### 4.3.1 Transaction Processing Engine (TPE)

The TPE is responsible for processing transactions in the system. The TPE operates like a normal relational database which stores schema information, allows for relationships and joins between entities and can also compute aggregate functions. The difference however is that the TPE does not store data. Rather, the TPE depends on the DMS to store data persistently. To execute a transaction, clients send requests directly to the TPE. The TPE requests for the required data items from the DMS. This is similar to the approach followed in [102] and [101]. The relationship between the TPE and the DMS can be implemented in two different ways. The first option is that each TPE can be responsible for a disjoint set of data on the DMS, in which case, each data item can only be accessed by the specific TPE assigned to it. In the second implementation, each TPE has access to all data on cluster such that the TPE acts as a transaction service to the DMS layer below. In this implementation, any TPE can have access to any data on any node in the DMS. As expected, the two implementations have different performance implications on the system. In the first approach, assigning TPEs to a disjoint set of

data would mean that a transaction that involves two or more data items assigned to different TPEs would span more than one TPE. In such cases, ACID transactions can only be implemented with two-phase commit protocol discussed in [section 2.4.1](#). All TPEs involved in the transaction would typically take part in the two-phase commit protocol. The TPE to which the transaction is submitted to, will act as the coordinator of the transaction. Two-phase commit protocol is an expensive process. In the second approach, using the TPE as a service would mean that the system can totally avoid two-phase commit protocol. Thus for each transaction, the TPE handling the transaction requests for the data items involved in the transaction from all the DMS nodes that stores each of the data. The transaction takes place in only one TPE. In both implementations, to improve performance, the TPE also stores information about the location of data on DMS. However, the evaluation of the difference in performance cost between these two approaches is beyond the scope of this thesis. The main functions of the TPE are summarized below:

- Receiving transactional requests from clients and managing such transactions
- Storing of schema information as NoSQL systems do not provide facilities for schema information
- Defining relationships between different entities of data
- Provide support for join operations as NoSQL do not support such operations

### **4.3.2 Data Management Store**

The Data Management Store (DMS) component represents the actual NoSQL cloud database such as MongoDB [86]. It stores all (Big) data persistently for the system. The DMS component is highly scalable in order to meet Big Data storage requirements. Further, it replicates data in terms of different replicas in order to ensure improved efficiency, high availability and fault tolerance. Replication is the common approach across all NoSQL systems. In the proposed system, the DMS layer, in collaboration with the Time stamp manager (TSM), implements the snapshot isolation protocol as a concurrency control mechanism for transactions in the system. When a transaction starts, the TPE requests the data items involved



in the transaction from the DMS. Before a data item is sent to the TPE, the DMS registers the transaction ID as well as the key of the data items at the Time stamp manager (TSM). The function of the TSM is to issue transaction timestamps. The DMS can then send the data to the TPE only after the start timestamp has been issued. The TPE can then perform updates on the data item. To commit data on the DMS, the DMS also uses the transaction ID registered with the TSM to retrieve a commit timestamp. If the TSM refuses to issue a commit timestamp, the transaction is then aborted and all changes made are rolled back. Whatever the case may be, the DMS notifies the TPE of the decision. The TPE then sends to relevant information as a response to the client.

In addition, the system introduces a new attribute called *lastModified*. Every data item stored on the DMS has an associated attribute called *lastModified*. This attribute stores the commit timestamp issued by the TSM for the most recently committed transaction on that data item. This is used to guarantee stronger consistency across replica and concurrent transaction execution explained in detail in [section 4.8](#). Since the TSM issues the commit timestamp, the TSM always has the most recent commit timestamp for each data item. A transaction can then use the information to confirm that it has read the latest version of that data item.

### **4.3.3 Time Stamp Manager**

The TSM component is central to managing consistency across nodes in the system. It manages the ordering and scheduling of transactions in the system. The TSM also interacts with DSM and TPE in order to schedule the execution of the different operations of a transaction. The TSM can also store transaction information in memory in order to reduce latency due disk I/O thus improving performance. Storing information in memory will allow the TSM process information much faster since it will reduce the need to perform a disk seek. This will improve the overall performance of the system as the TSM will be able to process transaction start and commit time faster. The TSM also keeps track of all active transactions. To do this, the TSM itself stores the transaction ID, key of data items involved in transactions as well as the start timestamps and commit

timestamps of each of these transactions. That way, the TSM is aware of the status of all ongoing transactions.

The main objective is to maintain consistency of data when concurrently accessed by different transactions. The proposed concurrency technique is to implement snapshot isolation which is non-blocking and provides higher concurrency and high efficiency in transactions processing. When a transaction request is made to the DMS, the DMS contacts the TSM for a start timestamp. Once granted, the TPE can begin to process transaction. After the transaction has completed, the DMS applies the changes and again requests for a commit timestamp from the TSM. If there has been any other transaction whose commit timestamp falls between the interval of the initial transactions start timestamp and commit timestamp and they both write or update the same data, the TSM refuses to issue a commit timestamp. Failure to issue a commit timestamp implies that there is a conflicting transaction and the on-going transaction must initiate an abort and rollback. This is how the First-writer-wins policy is implemented in this system. It is important to note the differences in read and writes when implementing Snapshot Isolation. If the committed transaction contains only read operations, then the on-going transaction need not abort.

Apart from issuing start time, TSM checks that a replica is up-to-date before it can grant that replica a transaction start time. The system employs an asynchronous model of replication. This means that replicas can be outdated for a very limited period; however, outdated replicas cannot be involved in transactions. Also, there is no notion of a master or primary replica in this system. Any replica can be involved in any transaction. The TSM in collaboration with the DMS has a mechanism for identifying out-of-date replicas. This process is explained in detail in [section 4.8](#).

#### **4.4 TRANSACTION STATE TRANSITION MODEL**

In order for a transaction to execute and access/modify a NoSQL database, it has to go through (or transition between) different states. This section explains the

process of state transitioning of transactions in the proposed model. This process helps the design of the execution of transaction protocols as well as their implementation, which are described in the next chapter.

As describe above, the proposed system is made up of three major components which include the TPE, DMS and TSM. The main duty of TPE is to process *NST* transactions on behalf of the system while the DMS serves as a stable storage for the database. The TSM works in collaboration with the other two systems to issue timestamps during transactions. It is necessary to add that the following constraints hold for NSTs that no two transactions that have a data item in common can have the same start timestamp or commit timestamp.

An active transaction that has not been aborted can be in one of the following four different states namely, (i) *initial* (ii) *pending* (iii) *applied* and (iv) *done*.

The transaction state is managed by the TPE. When a transaction is initiated by a client, the transaction state is set to *initial*. The *initial* state signifies that a transaction request has been initiated by a client. The transaction then proceeds to retrieve a start-time from the TSM. If it is successful, the transaction state is updated to *pending*. The (CRUD) operations are then carried out. If there is no failure in any part of the operation, the transaction state progresses to an *applied* state. Otherwise, the transaction is set to a *cancelling* state where rollback operation begins. A transaction in an *applied* state implies that all the enclosed operations in that transaction have been successfully executed. This does not however mean that the transaction would commit as there could be other conflicting transactions. Once a transaction state has been set to *applied*, the TPE will request for a commit-time from the TSM. If the commit-time is issued, the transaction state is set to *done*. At this point, a transaction is said to have committed and cannot be aborted. If a transaction is not issued a commit-time, the transaction is set to a *cancelling* state where rollback operation begins. As soon as the rollback operation is completed, the transaction state is set to *cancelled*. A transaction in a *cancelled* state is deemed aborted and cannot commit. The state diagram for a transaction is shown in Figure 4.2 below

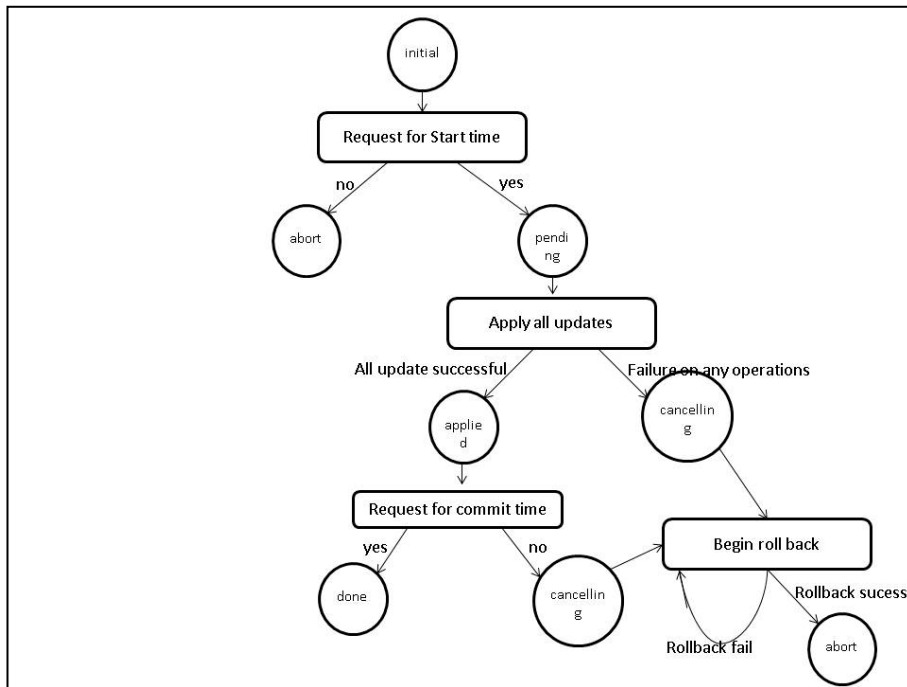


Figure 4.2: Transaction State Diagram

Under normal circumstances (without failures), a transaction state should follow the sequence below:

$$initial \rightarrow pending \rightarrow applied \rightarrow done \text{ (commit)}$$

However, there are a few scenarios which can cause a transaction to abort. They are explained in section 4.7.

#### 4.5 INTERACTION BETWEEN SYSTEM COMPONENTS

The three components of the proposed system interact with each other in order to coordinate different transactions. Figure 4.3 shows the various components and their interaction which each other. The TPE is the main component that carries out transactions in collaboration with the TSM. Each of these components has sub-components that carry out certain functions. The next section describes the functions of each of the sub-components in the TPE and the TSM.

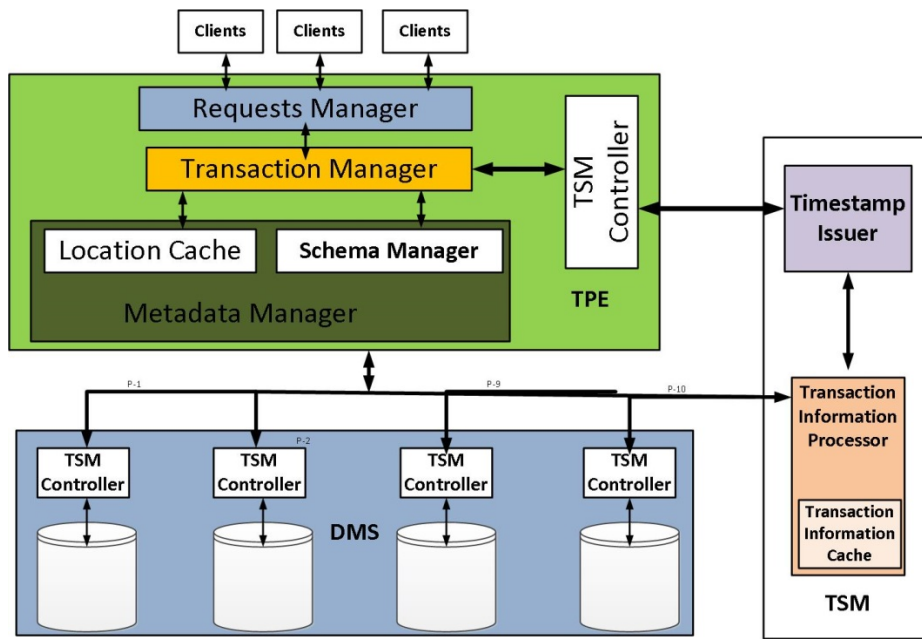


Figure 4.3: Component of Proposed System- NoSQL-TX

The TPE which manages transactions contains the following sub-components.

**Request Manager** - The request manager handles requests from clients. It provides an interface between client applications and the TPE. The request manager establishes a connection and maintains a session between applications and the transaction manager. The request manager also ensures that applications receive an acknowledgement or response from the TPE.

**Transaction Manager** - The transaction manager is an essential module in the TPE and is central to transaction processing. The transaction manager processes CRUD operations on the data items. It receives details of the operations to be performed from the request manager and generates a transaction ID for each operation. The transaction ID is a set of alphanumeric characters which forms a unique key used to identify each transaction known as the unique transaction identifier (UTID). The transaction manager also uses the transaction ID to manage the transaction states (explained in [section 4.4](#)). The transaction manager interacts with other components of the system to effectively perform transactions. For instance, before a transaction can operate on data item, the transaction manager requests the required data from the DMS. It stores data location information in the location cache. This way, the transaction manager can

know where each data item is located. The transaction manager also interacts with the schema manager to get information about the relationships that exists between the data entities. It is the duty of the transaction manager to ensure correctness in transaction processing.

**Meta-data manager** – The Metadata manager contains two sub-components which include the Location cache and the schema manager. The schema manager stores information about tables, relationships between the table entities specifies constraints that exist among the data objects. The schema manager also determines the level of permissions and accesses granted to data items. The location cache stores information about data location. This information is persisted on disk in the TPE but is also loaded into the location cache in memory to reduce disk I/O latency.

**TSM Controller** – the TSM manages interactions between components of the system (Transaction processing engine - TPE and Data management store - DMS) and the TSM. It guarantees that the components receive transaction start and commit timestamps issued by the TSM. The DMS and the TPE both contain TSM controllers as they both interact with the TSM. However, it is the DMS that makes requests for both start and commit timestamps from the TSM. But the TSM interacts with both the DMS and the TPE using the TSM controller.

The Time Stamp Manager (TSM) contains the following sub-components/

**Timestamp Issuer** - The main duty of the TSM is to issues timestamps. The timestamp issuer works as an issuing authority in collaboration with the Transaction Information Processor. The transaction information processor determines if the timestamp issuer will issue a timestamp or not.

**Transaction Information Processor** - The transaction information processor is a component of the timestamp manager whose function is to implement the snapshot isolation. The transaction information processor has a component which is called the transaction information cache. This is where it stores the transaction ID, transaction start and commit timestamps as well as the key identifier of data items involved in ongoing and recently committed transactions. Based on the stored information, the transaction information processor is able to identify any

conflicts in transactions. This in-turn determines if the Timestamp issuer would issue a commit timestamp or if the transaction should be aborted.

The diagram in Figure 4.4 shows communication and interaction between the different components of the proposed architecture. It depicts the flow of requests which are communicated between the client, TPE, DMS and TSM. Client represents user’s cloud application that submits transactions to the proposed system.

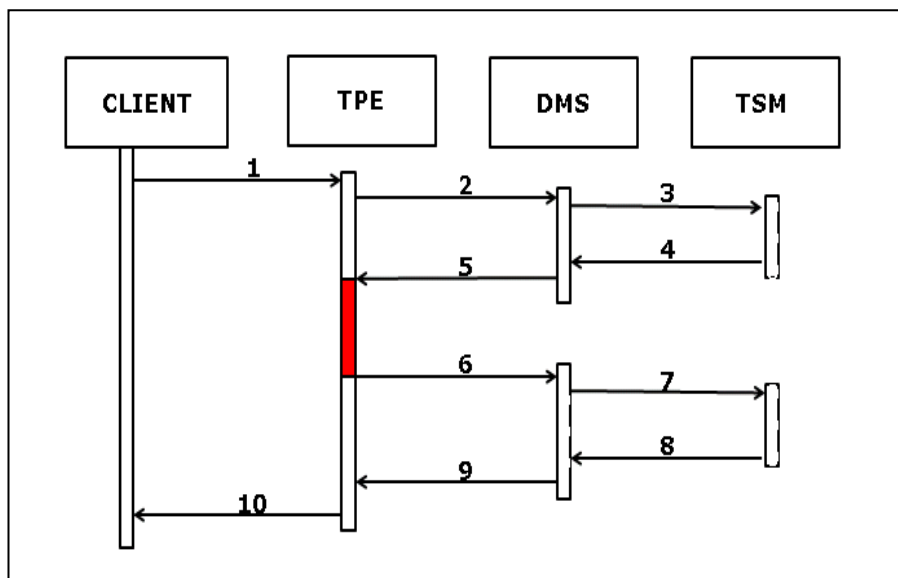


Figure 4.4: Interaction between Components of the System(NoSQL-TX)

Based on the above, the commit protocol for transactions in the proposed system is described as follows.

#### 4.6 COMMIT PROTOCOL

The different steps (see Figure 4.4) involved in the protocol are explained as follows.

1. A client initiates a request to start a new NoSQL transaction (*NST*). Recall, that *NST* is a set of begin, read,  $OP_1^r[DE]$ , write,  $OP_n^r[DE]$ , commit,  $Cmt_{i,}$ , and abort,  $Abt_{i,}$  operations (Section IV).

2. TPE receives client's request and generates an ID for the *NST* which is to be executed on NoSQL data. TPE then sends the *NST*'s ID and related information to the DMS.
3. DMS receives the *NST*'s ID and related information in order to know which data entities are to be accessed (read/updated) by the *NST*. DMS then sends the *NST*'s ID and related information about data entities to the TSM in order to ensure scheduling of *NST* and other transactions.
4. TSM saves the information about *NST* and it responds with a start-time of a transaction. This time serves as a start time-stamp, which is to determine the order of execution and also the commitment of the *NST*.

As in Section 4.1, if  $OP_i^r[DE], OP_i^w[DE] \in NST_i$ , then these read/write operations should be ordered either as  $OP_i^r[DE] < OP_i^w[DE]$  or  $OP_i^w[DE] < OP_i^r[DE]$ . That is, data entity, *DE*, should be read and written in a proper order following the time-stamp information.

5. Based on the above, DMS releases the required data entities to the TPE where *NST* is actually taken place. Note that the proposed architecture separates transaction processing from the actual NoSQL database system in order to ensure abstraction and transparency.
6. Once *NST* is completed, TPE sends the updates (made to data entities) to the DMS. This means that if *NST* updates a data entity (modify, delete) then DMS has to reflect this in the data store in order to ensure that data is consistent.
7. The DMS contacts the TSM to request a commit timestamp. The TSM checks if another transaction has updated the data after its start timestamp of the requesting transaction. If this happens, then the *NST* aborts and sends the information to the client through the TPE. Otherwise, it continues.
8. The TSM responds to the DMS with a commit timestamp. The DMS then stores the data in the data store.
9. The DMS responds with a commit message to the TPE. This means that *NST* is successfully committed using the commit operation,  $Cmt_i$ .



The algorithm below shows the process involved in issuing a commit time to a transaction  $T_i$

*getCommittime algorithm*

---

1. Send ID of  $T_i$  to TSM
2. **for each** data item  $\alpha$  in  $T_i$ , do:
3.     if  $\alpha$  exists in any Transaction  $T_j$  in TSMs record, do:
4.         If  $T_j$  *commit-time* between  $T_i$  *start-time* and  $T_{now}$  (where  $T_{now}$  is current-time)
5.             Send abort  $T_i$
6.             End Transaction  $T_i$
7.         Else
8.             commit  $T_i$
9.     Else
10.         commit  $T_i$
11. End

Note that the protocol above applies to transactions that have reached the *applied* state i.e. any transaction whose operations have all been executed to satisfy the atomicity property of transactions. Once the protocol is completed, the transaction state is changed to done.

#### 4.7 ABORT SCENARIOS

At each of the component, failures can occur at different stages of a transaction which can cause a transaction to abort. Three forms of aborts are defined below. The abort scenarios are as follows:

- Ab-S<sub>1</sub> – In this scenario, a transaction fails to collect a start timestamp. This can happen as a result of a connection failure, an error at the TPE or some other error. This then leads to a situation whereby the TSM is unable to issue a start time. In this scenario, the transaction state follows the sequence (see State Transition in, Figure 4.2).

*initial* → (abort)

- Ab-S<sub>2</sub> – This is a scenario when any part of a transaction fails or does not complete. This can only happen at the TPE. There are several possible causes such as a problem or error at the TSM, or the data not being reachable on the DMS (see State Transition, Figure 4.2).

*initial* → *pending* → *cancelling (rollback)* → *cancelled (abort)*

- Ab-S<sub>3</sub> – This is scenario when a transaction fails to collect a commit timestamp. Like in Ab-S<sub>1</sub>, this can also be as a result of a connection failure, an error at the TPE or some other error. All such errors can prevent the TSM from issuing a start time. In addition, scenario Ab-S<sub>3</sub> is also caused by conflicting transactions. In the transaction state diagram (see Figure 4.2), the path followed is described below.

*initial* → *pending* → *applied* → *cancelling (rollback)* → *cancelled (abort)*

Besides the above stated abort scenarios, the system assumes a failure free environment. This work does not consider system or network failures. When an exception or failure occurs, the transaction halts and commences a rollback which ultimately leads to an abort. An exception is an event that causes a failure thereby forcing a transaction to abort. It can be anything from the failure of the TSM to issue a start-time or commit-time to conflicting transactions, which lead to an abort. Recall from section 4.1, “a transaction must end in a commit or an abort i.e. a transaction follows the sequence (*Begin* | *OP<sub>i</sub>* | *Cmt* | *Abt*) but with the condition that either *Cmt* (commit) or abort (*Abt*) occurs only once within the sequence (where *OP<sub>i</sub>* is a set of operations that can occur during the transaction)”. A rollback operation would commence if a transaction is aborted mid-way so as to preserve consistency. However, the rollback operation will be pre-determined by the state of the transactions when the abort took place.

The next section explains how the proposed system uses the TSM to maintain consistency among replicas in the presence of concurrent transactions using the TSM

#### 4.8 A PROTOCOL FOR MANAGING TRANSACTIONS ACROSS ASYNCHRONOUS DATA REPLICATION

Replication is a commonly used technique used to guarantee high availability in NoSQL databases. As explained in [section 2.6.3](#), replication is a process of maintaining multiple copies of a database in different locations to provide fault tolerance and guarantee higher levels of availability. In asynchronous data replication, replicas are updated at a later time after the transaction is committed. This means that a replica can have outdated data for a short period of time.

This thesis has introduced a new model for managing consistency across replicas using the TSM. In this section, a protocol for ensuring that only up-to-date replicas are involved in transactions to guarantee consistency is explained. The goals of replication in NoSQL databases include:

**Availability** - Replication increases availability in that during node failures or network partitions, other replicas can still continue to process read and write requests.

**Read / Write latency** - Many NoSQL systems such as PNUTS, Spanner use wide area (geographic) replication in order to lower response times. When requests are made, the replica that is closer to the client responds to its request. This is used to guarantee lower latencies.

**Scalability** - Replication is also used to balance load across multiple nodes (of NoSQL databases) such that each node is not under heavy traffic. When the number of requests on a replica increases, the requests exceeding the capacity can be directed to other replicas.

**Fault tolerance and Data persistence** - Replication guarantees that data is not lost during failures by providing multiple copies of the same data. Fault tolerance allows a system to continue operating as normal in the event of a failure.

Therefore, if one server is down, replication ensures that there one or more other servers that can still respond to user requests.

However, implementing replication introduces the challenge of maintaining consistency among replicas. Generally replication is classified into two main categories which include Lazy and Eager replication. These are explained in [section 2.6.3](#). Implementing Eager replication is non-trivial and as such most systems implement various forms of lazy replication. However, implementing lazy replication has breached consistency guarantees. This is because Lazy replication will allow transactions to see stale data versions [49]. As a result, most NoSQL databases use some form of quorum based replication. In such replication, an agreed number of replicas (usually less than the total number of replica) can accept read and write requests. Also, to reduce bandwidth traffic, most replicated environments makes use of a master-slave replication which is also known as primary – secondary replication.

This section explains how the proposed system achieves consistency across transactions. The proposed system implements an asynchronous (lazy) replication model. However, unlike most systems, in the proposed model, the notion of a primary or master replica does not exist. In the proposed model, any server can attend to requests such as reads and writes. The system depends on the TSM to identify replicas that are out of date and prevents transactions from accessing such replicas. As part of the protocol, the proposed system enforces the following constraints and conditions:

1. Recall the constraint of the proposed system in section 4.1

“No two transactions can have the same commit time”

Therefore, the constraint below applies such that

$$\text{If } \{DE_{\alpha} \in T_a \text{ and } DE_{\alpha} \in T_b\}, \text{ and } \left( \{T_{a \text{ commit}} < T_{b \text{ commit}} \cdot T_{a \text{ commit}} > T_{b \text{ start}}\} \mid \{T_{b \text{ commit}} < T_{a \text{ commit}} \cdot T_{b \text{ commit}} > T_{a \text{ start}}\} \right)$$

Then  $T_{a \text{ commit}} \neq T_{b \text{ commit}}$

(Where  $DE_{\alpha}$  is a data entity  $\alpha$ ,  $(\cdot)$  represents ‘and’,  $(\mid)$  represents ‘or’)

The expression above implies that if two concurrent transactions  $T_a$  and  $T_b$  operate on the same data item  $DE_\alpha$  and are in conflict, then  $T_a$  cannot have the same commit timestamp as  $T_b$ . This constraint is used to enforce ordering across concurrent transactions.

2. The *lastModified* attribute (explained briefly in [section 4.3.2](#)) for each data item is the same across all replicas and is equal to the last commit timestamp issued by the TSM for the most recent transaction committed on the data item. If a data entity  $DE_\alpha$  is replicated on three servers A, B and C, then the *lastModified* attribute (which is a commit timestamp) of  $DE_\alpha$  should be the same across the three servers A, B and C. Therefore  $lastModified_{\alpha A} = lastModified_{\alpha B} = lastModified_{\alpha C}$

(Where  $lastModified_{\alpha A}$  is the '*lastModified*' attribute of data entity  $DE_\alpha$  on server A)

3. If the commit timestamp on a data item,  $\alpha$ , from a replica A is different from the commit timestamp for the last transaction on  $\alpha$  at the TSM, then either replica A is outdated or another transaction as committed on replica A

The steps involved in the protocol are explained as follows

1. A transaction retrieves a commit timestamp from the TSM before it commits. This commit timestamp is stored in the *lastModified* attribute of all the data entities involved in the transaction.
2. The commit timestamp is sent along with the updated data to all the replica servers of the data entities after the transaction has committed on the server involved in the transaction. This is asynchronous replication.
3. The commit timestamp is also stored in the *lastModified* attribute of each of the replica servers. Therefore, the *lastModified* attribute for a particular data entity should always contain the same timestamp across all replicas. This is because the timestamp that is saved in the *lastModified* attribute is not the time which the update reaches the replica but the commit time issued by the TSM to the replica that sends the transaction request.

4. When a new transaction is initiated, the TPE consults with the TSM to verify that the data items from the DMS are not outdated and that no other transaction has committed on another replica of any of those data items. It can verify this by comparing the *lastModified* attribute with the most recent commit timestamps issued by the TSM for each of the data entities.
5. If the *lastModified* attribute is not the same time as the last commit time issued by the TSM for each of the data entities, the TPE knows that the data is outdated. This way, the TPE can guarantee the consistency of the data items involved in the transaction.

By following these steps, transactions can identify stale replicas and transaction requests are redirected to the most current replica.

#### 4.9 SUMMARY

This chapter explained the approach (which is referred to as NoSQL-TX) of the proposed system to implement transactions in NoSQL cloud databases. The theoretical model of the NoSQL-TX is also defined in this chapter. The approach makes use of snapshot isolation as a concurrency control technique. This means that the system avoids the overhead involved in locking data and improves availability. Snapshot isolation also helps to avoid anomalies such as dirty reads, phantom reads and non-repeatable reads.

The architecture of NoSQL-TX was described in this chapter as a loosely coupled architecture with three components which include the Data management Store, Transaction Processing Engine and the Time Stamp Manager. Each of these components performs certain roles that are critical to the health of the system.

An ongoing transaction passes through four different phases during its life time. They include: *initial*, *pending*, *applied* and *done* phases. The events that trigger a transaction to change from one phase to another were explained in this chapter. When an abort occurs, the transaction moves to a *cancelling* phase. During this

phase, all the changes made are rolled back before the transaction state moves to a *cancelled* phase.

To execute an operation successfully, all the components of the system must interact with each other. The protocol followed by their interaction is explained in this chapter.

Finally, the chapter explains the protocol for managing consistency when the data in the system is replicated. Most cloud databases make use of replication to improve availability. This chapter introduces a new protocol for guaranteeing consistency among replicas.

The next chapter explains in detail, the implementation of the proposed system.

# CHAPTER 5

## IMPLEMENTATION OF THE NoSQL-TX SYSTEM

This chapter discusses the implementation of the proposed system as a prototype. Section 5.1 outlines and explains the design objectives of the proposed NoSQL-TX system. The tools and technologies used in implementing the system are described in section 5.2. Section 5.3 explains the different types of operations supported by the system and the algorithm of each of these operations. Section 5.4 explains the application domain used to implement and test the proposed system.

### 5.1 DESIGN OBJECTIVES

In line with requirements of transactional systems highlighted in [119], the primary requirement for design of the prototype system is to provide transactional support which guarantees that ACID properties are preserved in NoSQL databases. The design also put performance metrics into consideration. As such, the following characteristics represent non-functional requirements of the prototype system.

- High-throughput which reflects a high rate of successful transactions per unit time.
- High concurrency that do not violate the consistency and isolation properties of transactions.
- Low latency and shorter response times in responding to client request.

These design objectives are in line with objectives II and III of this thesis outlined in section 1.4 which include:

- Design a new framework for transaction management in NoSQL databases



- Develop and implement the proposed framework as a prototype system using cloud data management tools and technologies

### 5.2 IMPLEMENTATION TOOLS AND TECHNOLOGIES

The prototype implementation makes use of different tools and technologies in order to implement the various components of the proposed system such as TPE, TSM, DMS, etc. The rationale for the choices of the various tools and technologies are explained and justified as follows:

#### MongoDB –NoSQL Database:

The proposed system uses MongoDB to implement the cloud storage part or the DMS layer. MongoDB is a NoSQL database that belongs to the document family of NoSQL Databases (see [section 3.2](#)). It uses JavaScript Object Notation (JSON) as its implementation language. MongoDB does not support multi-key transactions.

The prototype system can be implemented using other NoSQL databases. This research chooses MongoDB for the implementation of some of the components of the proposed system due to the following reasons:

First MongoDB is widely used in real applications and in industry. For instance Ebay uses MongoDB to store metadata for all items advertised on their website<sup>2</sup>. The UK Met office also uses MongoDB for storing climate data used in weather forecasts<sup>3</sup>.

Second, the JSON notation of MongoDB allows relationships amongst data entities to be expressed. However, MongoDB does not enforce this relationship i.e. it has a flexible schema which is a key characteristic of NoSQL databases. Third, MongoDB databases have a relatively higher speed (for simple operations) when compared with relational databases [120]. This will help to achieve low latency for operations. This is in line with the design objectives set out for the proposed system.

---

<sup>2</sup> <https://www.mongodb.com/industries/retail>

<sup>3</sup> <https://www.mongodb.com/industries/government>

## IMPLEMENTATION OF THE NoSQL-TX SYSTEM

### Programming Language:

The proposed system uses Python as a programming language in order to implement the proposed system using the middleware architectural approach. In the implementation, the TPE layer code is written in python language. The code in the python modules defines a programming abstraction that implements the different operations of a transaction (which are explained in [section 5.3.1](#)) supported by the prototype. The python IDLE version used is Python 2.7<sup>4</sup>. MongoDB is written in JSON, therefore, to access the MongoDB database, the pymongo api<sup>5</sup> was installed. Therefore JSON scripts used to query the MongoDB can be embedded into the python programming.

### SQLite Database System:

The TSM is implemented using a combination of python and a lightweight relational database, SQLite. SQLite can store information in memory. This makes it faster and easier in processing transactions.

The python code implements the various constraints and interaction between the TSM and the other components. It also generates and issues the transaction timestamps. The lightweight relational database serves as a storage system for the TSM. It stores transactional information of on-going and recently completed transactions such as the transaction timestamps, transaction IDs and data items involved in transactions. The lightweight database used is SQLite which can store information in memory. This makes it faster and easier for processing transactions. To access SQLite from python 2.7, the sqlite3 API<sup>6</sup> was installed. To update the database, SQL scripts are embedded in python.

### Computer System and Hardware Specification:

The DMS storage layer is implemented on an 8GB RAM Linux Ubuntu system with three replicas. The replicas are hosted on Ubuntu Juju<sup>7</sup> which is a cloud hosting Linux platform used to deploy, configure, manage, and scale cloud services on

---

<sup>4</sup> <https://www.python.org/download/releases/2.7/>

<sup>5</sup> <https://api.mongodb.com/python/current/>

<sup>6</sup> <https://docs.python.org/2/library/sqlite3.html>

<sup>7</sup> <https://jujucharms.com/docs/1.24/about-juju>

## IMPLEMENTATION OF THE NoSQL-TX SYSTEM

physical hardware. The nodes are connected via Ethernet to a MAAS (Metal as a Service<sup>8</sup>) cluster and managed by a cluster master in a local area network. See Figure 5.1 for the cluster controller (master) configuration. Figure 5.2 shows the nodes in the cluster and their addresses managed in Ubuntu. Figure 5.3 shows the configuration of one of the nodes (cgfk6.maas) in the cluster. Figure 5.4 shows a putty connection to the MongoDB service running in Juju.

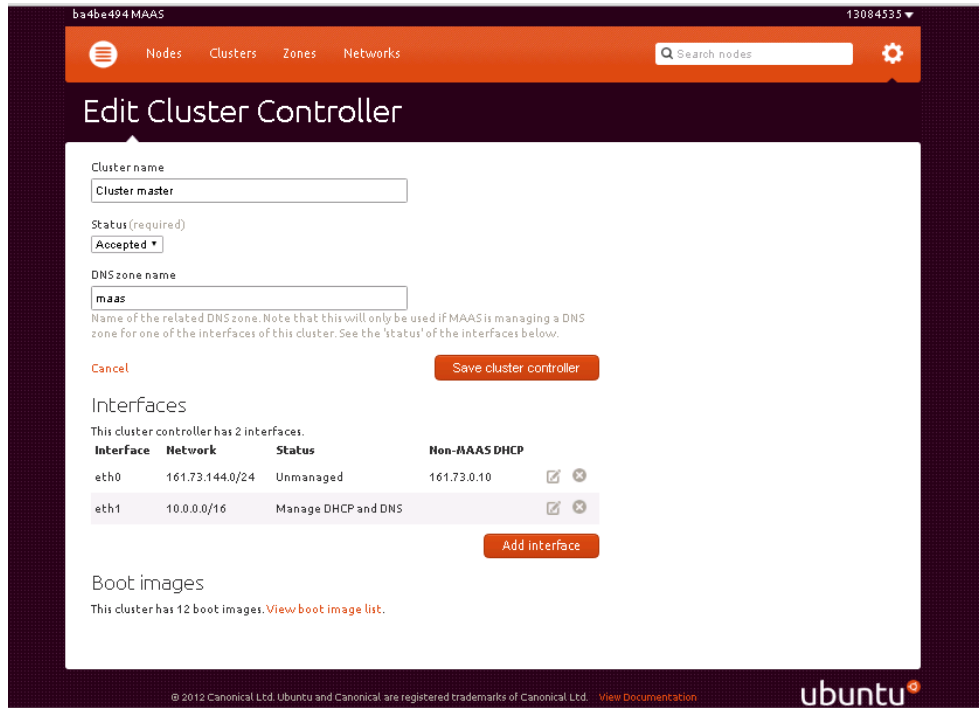


Figure 5.1: MAAS Head Controller Configuration

<sup>8</sup> <http://www.ubuntu.com/cloud/maas>

## IMPLEMENTATION OF THE NoSQL-TX SYSTEM

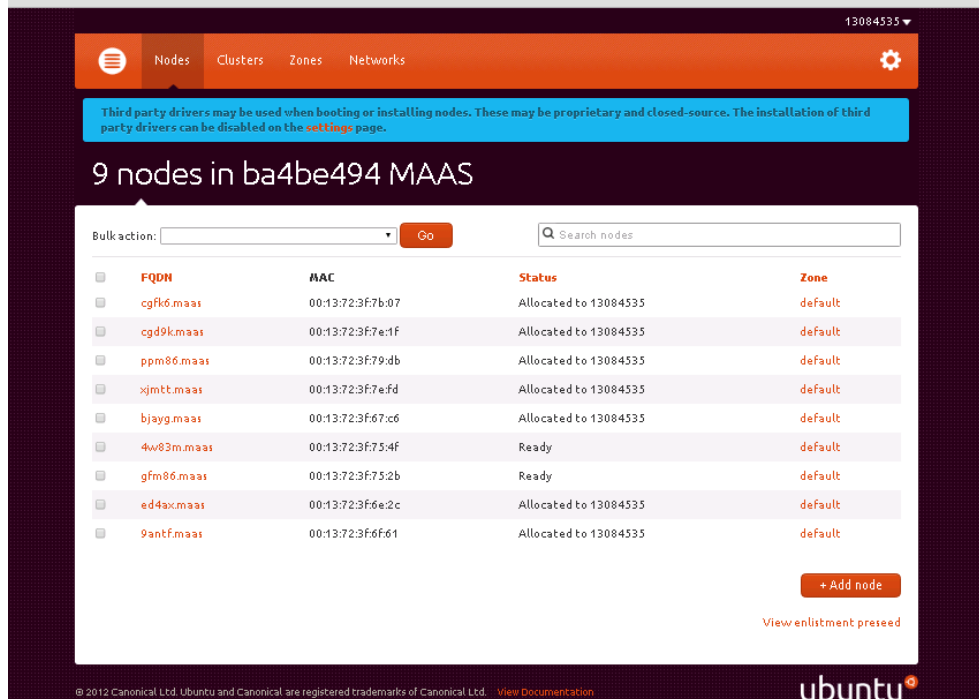


Figure 5.2: Nodes in the cluster with their local addresses

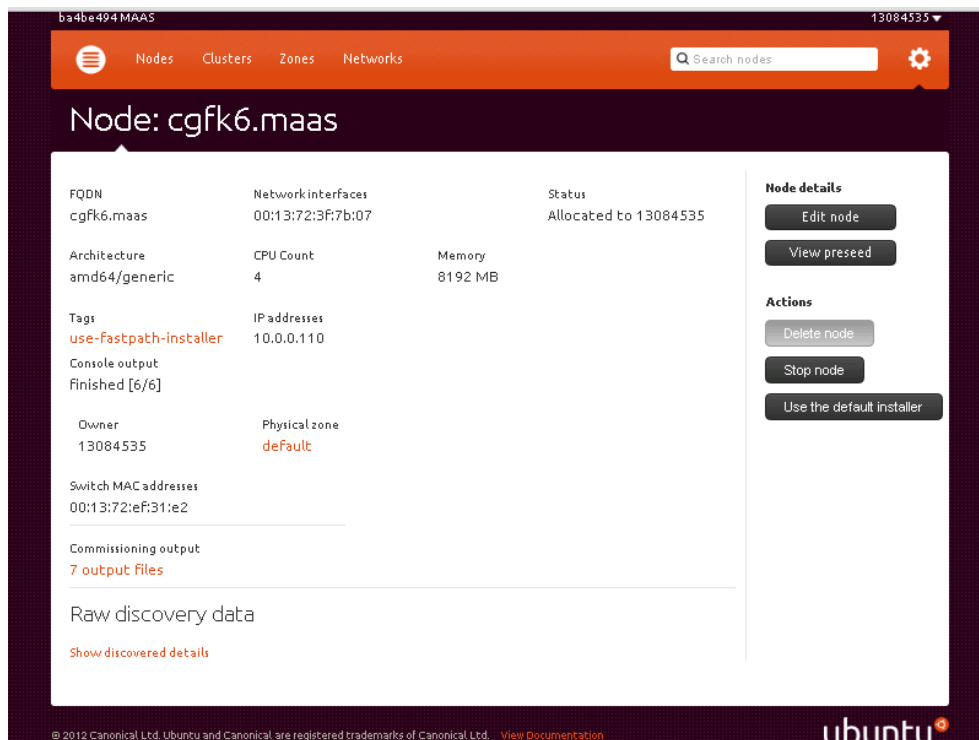


Figure 5.3: Configuration of One of the Nodes - Address 10.0.0.110

## IMPLEMENTATION OF THE NoSQL-TX SYSTEM

```
1004515@b4b4e64:~$ juju status shard1
juju@b4b4e64:~/home/130845352_juju$ status shard1
environment: maas_adevoic
machine0:
  *?
  agent-state: started
  agent-version: 1.20.14
  dns-name: ed4ax.maas
  instance-id: /MAAS/api/1.0/nodes/node-54172400-2701-11e2-0019-0019724be494/
  series: trusty
  hardware: arch=amd64 cpu=cores=4 mem=0192M tags=use-2acpath-installer
services:
  shard1:
    charm: cs:trusty/mongodb-27
    can-upgrade-to: cs:trusty/mongodb-37
    exposed: true
    service-status: {}
    relations:
      replica-set:
        - shard1
    units:
      shard1/0:
        workload-status: {}
        agent-status: {}
        agent-state: started
        agent-version: 1.20.14
        machine: *?
        open-ports:
          - 27017/tcp
          - 27019/tcp
          - 27021/tcp
          - 20017/tcp
        public-address: ed4ax.maas
```

Figure 5.4: MongoDB service running via Putty

The compute nodes on which the TSM and TPE are running have 16GB RAM and Intel i7 3.40GHz processors. The diagram below shows the component set-up.

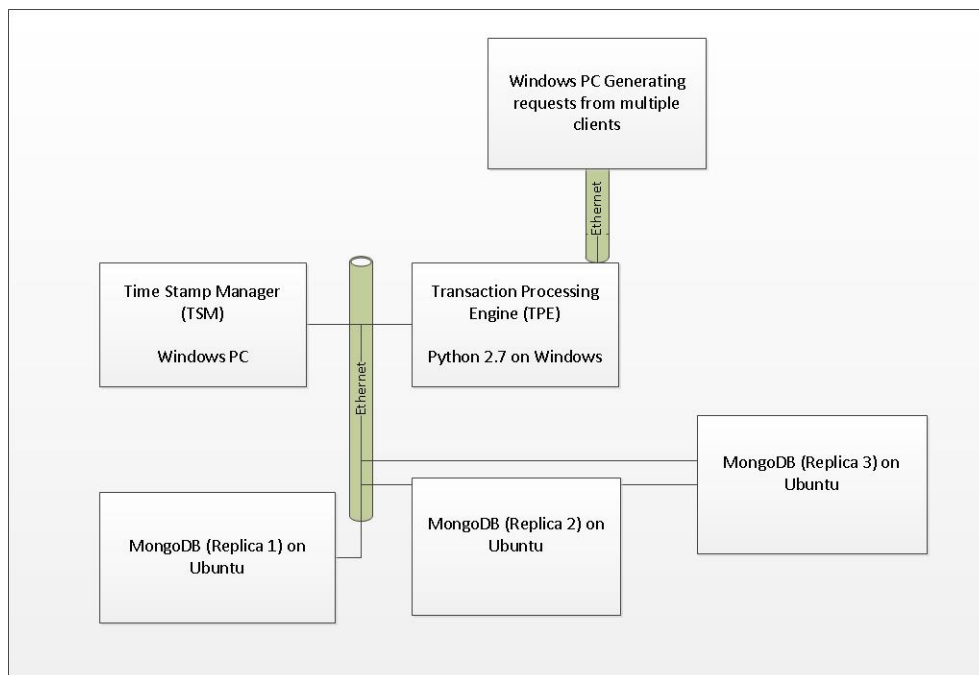


Figure 5.5: Hardware Setup of Proposed System

### 5.3 IMPLEMENTATION OF TRANSACTION OPERATIONS

This section explains the set of operations which are implemented (and supported) – by the prototype system. The decision to implement snapshot

isolation using an entirely different component (the TSM) gives the system an advantage of being able to centrally monitor the execution of transactions across the system. This design also helps the system to easily identify all on-going transactions. The TSM can know this by checking all transactions that have not been issued a commit time. This feature is very strategic to the system as it allows the system to introduce two new types of operations namely (i) read-latest and (ii) update-latest. These two operations provide stronger consistency guarantees. Also, operations executed in this mode are serializable. The operations supported by the proposed system are explained below through various algorithms.

### **5.3.1 Types of Operations**

In general, the operations supported by relational databases include Create, Read, Update and Delete (CRUD). The 'Create' operation also known as an 'insert' adds one or more record to a database table. Read operation is a 'select' operation which retrieves the result of a query from the database. An Update operation changes one or more existing record in a database. Delete operation removes an existing record from a database. NoSQL databases support mainly 'get' and 'put' operations. A 'get' operation is equivalent to a 'read' while a 'put' operation can be a 'create' and 'update'. The operations supported by the prototype system are explained below.

#### **5.3.1.1 Read**

The read operation which is equivalent to a read in CRUD is a simple 'get' operation. The TPE simply requests for a data item from the DMS using the key identifier of the required data item. The DMS responds with the data to the TPE. Read transactions may not reflect updates from on-going transactions. Since there is no locking, data is always available but it may not always be consistent. This means that a read operation does not put into consideration ongoing transactions. As such, a read operation would only reflect any data item that has

been committed at the time of the operation. Below is the algorithm for a read operation.

---

#### Algorithm Read

---

1. Function **READ** key
2. Generate UTID() (at TPE)
3. Send (UTID, key) to DMS
4. Value → Return (key, data\_item)
5. Return Value
6. End function

An example of a read operation is retrieving a customer record from a banking application. The user supplies an account ID which the system uses to identify the record in the database. The Figure 5.6 shows a snippet of the read operation.

```

32
33 def readOp():
34
35     try:
36
37         account = random.choice(test)
38         calSTime = datetime.datetime.utcnow()
39         print(calSTime)
40         #f.write(repr(calSTime) + ",")
41         result = db.accounts.find_one({"_id" : account})
42         if result is None:
43             f.write("\n " + "Fail,")
44         else:
45             f.write("\n" + repr(account) + "," + repr(result['balance']) + ",")
46     except ValueError as err:
47         print(err)

```

Figure 5.6: Read Operation Codes in Python

### 5.3.1.2 Read-latest

This research introduces a new type of read operation called the read-latest in order to guarantee stronger consistency. The read-latest operation is also a form of read operation albeit more complex than the simple read operation. The read-latest operation aims to get the most recently committed value of requested data and puts on-going transactions into consideration. Most concurrency control techniques do not put into consideration ongoing transactions. Even in Locking, systems still allow locked data, which may be stale due to ongoing transactions, to be read. Before it responds to a request, a read-latest transaction checks the TSM to ensure that there is no on-going transaction for that data item. The TPE sends a request to the DMS. The DMS then contacts the TSM to confirm that there is no on-going transaction on the data item requested. Recall from [section 4.5](#) that the

## IMPLEMENTATION OF THE NoSQL-TX SYSTEM

TSM stores information on all running transaction and is able to monitor the activities of each transaction. If there is none, the DMS responds to the TPE with the data. If on the other hand there is an on-going transaction on the data item, the DMS waits for the transaction to complete before responding to the TPE. The read-latest operation may incur a slightly higher latency but it guarantees a stronger level of consistency for read operations. To prevent unnecessary waits, there is a maximum (configurable) upper bound time limit for every read-latest operation. Once this is reached, the transaction would timeout and the DMS must respond with a simple read-transaction or set the data unavailable in which case the client can request a simple read operation. A read-latest operation will incur a slightly higher latency than a read operation even if there are no on-going transactions. This is because the read-latest operation makes an extra journey to the TSM. To optimize the process of executing this operation, the TPE can communicate directly with the TSM through its TSM controller. It can do this simultaneously while requesting for the needed data from the DMS.

Using the earlier example, to retrieve an account, a user supplies the user account ID. The transaction uses the ID to verify from the TSM if there is any ongoing transaction on the user supplied ID. If there is, the transaction waits for a specified amount of time. Otherwise, it returns the account details.



---

*Read-latest algorithm*

---

1. Function **READ-LATEST** key
2. Generate UTID() for  $T_i$
3. Send key to DMS
4. **For each** data item  $\alpha$  in a read  $T_i$ , do:
  5. **If**  $\alpha$  exists in any on-going transaction  $T_j$  in TSMs record, do:
    6. Start *counter*
    7. while *counter* < upper-bound time
    8. wait for all  $T_j$  to commit
    9. **if** *counter* >= upper-bound
    10. respond  $\alpha$  is unavailable
    11. exit
    12. else
    13. **once** all  $T_j$  committed, respond with latest value for  $\alpha$
  14. **else**
  15. respond with latest value for  $\alpha$
16. End Function

Below is a snippet of the code that implements the read-latest operation.

```

175 f = open('readlatestthirtywoolinst1.csv', 'w')
176 started = time.clock()
177 account = random.choice(test)
178 calSTime = datetime.datetime.utcnow()
179 stTime = str(calSTime)
180 #print(calSTime)
181 f.write("\n" + "ReadLatest," + repr(account) + "," + repr(stTime) + ",")
182 conn = sqlite3.connect('pach')
183 cursor = conn.execute("select * from SNAPSHOT where ACCOUNT_ID = ? AND T_COMMIT ISNULL", (account,))
184 result = (cursor.fetchone())
185 while (time.clock() - started < 2 and result is not None) or (time.clock() - started < 2 and result == (None,)):
186     #if result != None:
187     cursor = conn.execute("select * from SNAPSHOT where ACCOUNT_ID = ? AND T_COMMIT ISNULL", (account,))
188     result = (cursor.fetchone())
189     print(result)
190     time.sleep(1)
191
192 if time.clock() - started > 2:# or result is None or result == (None,):
193     print("Transaction taking too long - Failed")
194     f.write("Fail, Data Item not Available")
195 else:
196     findlatest = db.accounts.find_one({"_id": account})
197     thebalance = str(findlatest['Balance'])
198     f.write(repr(thebalance) + ",")
199     #print(calSTime)
200     print(account, findLatest['Balance'])
201

```

Figure 5.7: Read-latest operation

### 5.3.1.3 Write-New

Write-New operation is equivalent to *create* in the CRUD operations of a database. It is a simple write issued to the DMS by the TPE. Since NoSQL

databases are schema-less, inserting a new record will automatically succeed even if the table (or collection as the case may be) did not initially exist. A write-new operation does not necessarily need to contact the TSM. The client sends the request to the TPE and the TPE sends that data item to the DMS.

---

### Write-New algorithm

---

1. Function WRITE-NEW (*key*, *data-item*)
2. Generate UTID() (at TPE)
3. Send (*key*, *data-item*) to DMS
4. Return (Ack)
5. End Function

#### **5.3.1.4 Update**

The update operation is a write operation that changes the value of an existing data item. For an update to succeed, the data item must exist already. An update operation starts with a read of the existing value of the data item. Therefore, an update operation takes the key and the data item (provided by the client) as arguments. The TPE requests for the data item from the DMS using the key to identify the requested data item. The DMS then sends a start-time request to the TSM. The TSM responds with the start time and the DMS then sends the data item to the TPE, where the update takes place. The updates are then sent to the DMS. As explained earlier, when an update takes place, and before it can commit, the DMS must check with the TSM that no other transaction has occurred on that key item. Otherwise the transaction aborts. Once the TSM guarantees that there are no conflicting transactions, the TSM sends the commit timestamp. The data is committed and an acknowledgement message is sent to the TPE. Below is the algorithm for the update operation.

---

*Update algorithm*


---

1. Function **UPDATE** (key, data-item)
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items) to DMS and TSM
4. DMS sends the UTID to the TSM and gets a start-time
5. DMS releases the data-item to the TPE
6. TPE performs all the updates and sends back to DMS
7. DMS applies update and proceeds to TSM
8. For each data item  $\alpha$  in update  $T_i$ , do:
  9. If  $\alpha$  exists in any on-going transaction  $T_j$  in TSMs record, do:
    10. If  $T_j$  *commit-time* between  $T_i$  *start-time* and  $T_{now}$  (where  $T_{now}$  is current-time)
      11. Send abort  $T_j$
      12. End Transaction
    13. Else
      14. Issue  $T_j$  *commit-time*
      15. Commit  $T_j$
    16. Else
      17. Commit  $T_j$
18. End Function

Using the same banking application as an example, to perform an Update operation, a user supplies the account ID of the account to be updated. The system performs a read operation described in [section 5.3.1.1](#) to retrieve the data. The user then supplies the update information. The system then performs a commit using the protocol explained in [section 4.6](#).

### **5.3.1.5 Update-Latest**

Like the read-latest, the update-latest is also a new type of operation that has been implemented in the proposed system. The update-latest provides a stronger consistency guarantee and it reduces the probability that a write operation would abort. The TPE issues a write request with the key of the data-item to be updated. The DMS checks that the data item exists and then reads that data item. It then consults the TSM to know if there is any on-going transaction on that key-item. If

there is no transaction, the DMS responds to the TPE with the data and the TPE performs the necessary update and sends it to the DMS. The DMS proceeds to commit the data using the commit protocol highlighted in [section 4.6](#). If on the other hand, there is an on-going transaction, again, the DMS must wait for the transaction to complete before the TSM issues a transaction start-time. Once a transaction start-time is issued, the DMS can then proceed to send the requested data item to the TPE. As in the read-latest operation, the proposed system sets a maximum (configurable) upper bound time limit for every update-latest operation in order to prevent unnecessary waits. Once this time is reached, the DMS must respond with a 'data unavailable' message. Again, an update-latest operation may also incur a higher latency than an update operation even if there are no on-going transactions. This is because of the extra time it may take to wait for on-going transactions to commit.

---

*Update Latest algorithm*


---

1. Function **UPDATE-LATEST** (key, data-item)
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items) to DMS and TSM
4. DMS sends the UTID to the TSM to retrieve a start-time
5. **For each** data item  $\alpha$  in a read  $T_i$ , do:
  6.       If  $\alpha$  exists in any on-going transaction  $T_j$  in TSMs record, do:
    7.           Start *counter*
    8.           **while** *counter* < upper-bound time
      9.               wait for all  $T_j$  to commit
      10.              **if** *counter* >= upper-bound
        11.                 respond  $\alpha$  is unavailable
        12.                 exit
      13.              **else**
        14.                 DMS releases the data-item to the TPE
        15.                 TPE performs all the updates and sends back to DMS
        16.                 DMS applies update and proceeds to TSM
        17.                 For each data item  $\alpha$  in update  $T_i$ , do:
          18.                   **If**  $T_j$  *commit-time* between  $T_i$  *start-time* and  $T_{now}$   
(where  $T_{now}$  is current-time)
            19.                     Send abort  $T_j$
            20.                     End Transaction
          21.                    **Else**
            22.                     **Commit**  $T_j$
        23.                 **Else**
          24.                    **Commit**  $T_i$
  25. **End Function**

### 5.3.1.6 Multi Key Transactions

A Multi-key transaction is a transaction that involves more than one data key item. As discussed extensively in chapter 2, NoSQL databases, because of their simple data model (de-normalized data model), do not support multi-key transactions. Thus, because of the lack of support for table joins, they do not support multi-key transactions. Note that joins essentially involves multiple keys from one or more tables. This shows that the support for multi-key transactions

represents a novel contribution of this work. The multi-key operation follows the pattern of the update operation but involves an exchange of information between two or more data items such as a transfer of funds from one account to another. To perform this operation, a user follows the scenario 1 given in [section 2.2](#). A user supplies his account ID, the account ID of another user and the amount to be transferred. The system performs a read operation on both accounts and applies an update operation by deducting the amount from one account and adding it to another account. The system also verifies that sum of both accounts before the transaction is equal to the sum of both accounts after the transaction.

Recall, from [section 4.3.1](#) and [section 4.5](#), the TPE stores schema information in the schema manager. Therefore, the TPE understands the application logic and the relationships that exist between the entities data stored at the DMS. Therefore the system can support applications to perform operations among multiple keys. The multi-key operation pseudocode is shown below.

---

*Multi-Key algorithm*

---

1. Function **Multi-Key** (key, data-item) 1..n
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items 1..n) to DMS and TSM
4. DMS sends the UTID to the TSM and gets a start-time
5. DMS releases the data-item(1..n) to the TPE
6. TPE performs all the updates and sends back to DMS
7. DMS applies update and proceeds to TSM
8. For each data item  $\alpha$  in multi-key  $T_i$ , do:
  9. If  $\alpha$  exists in any on-going transaction  $T_j$  in TSMs record, do:
    10. If  $T_j$  *commit-time* between  $T_i$  *start-time* and  $T_{now}$  (where  $T_{now}$  is current-time)
      11. Send abort  $T_j$
      12. End Transaction
    13. Else
      14. Commit  $T_j$
    15. Else
      16. Commit  $T_j$
17. End Function

As can be seen from these operations, the system considers consistency as a very important aspect of transactions that cannot be violated. As such there are consistency checks on most types of writes. The system relies on the performance of the TSM to perform operations. The multi-key operation is a novel operation that most NoSQL databases do not support. The operation allows applications logic to be expressed at the level of the database and also ensures that the consistency property of the database is preserved. The protocol explained in [section 4.8](#) guarantee that stale replicas are not involved in transactions.

The next section explains the various scenarios that can cause a transaction to abort and how the system handles aborts to preserve consistency.

### **5.3.2 Aborts Scenarios for Operation**

Recall that from the definition of snapshot isolation ([section 4.2.1](#)), that it does not block operations of a transaction. Furthermore, read operations would always be successful, i.e., read operations would always return a value. However, as an exception, the read-latest operation introduced in this model may not necessarily return a value due to its consistency guarantee. For each of the operations described in [section 5.3.1](#), an abort protocol can be triggered. [Section 4.7](#) explains three types of scenarios which can lead to aborts. This section would explain the algorithm that each of the operations (with the exception of read operations) would follow to implement any of the three types of aborts. A brief description of the abort scenarios is reiterated as follows.

Ab-S<sub>1</sub> – Occurs when a transaction is unable to retrieve a start timestamp.

Ab-S<sub>2</sub> – Occurs when one part of the operations in a transaction fails thus violating the atomic properties of transactions

Ab-S<sub>3</sub> – Refers to when a transaction fails to get a commit timestamp from the TSM.

The algorithms to implement the above mentioned scenarios in each of the operations are explained in the next section. The Read-Latest and Write-New

operations do not implement any of the three abort scenarios because they do not require a start or commit timestamp in their execution.

### 5.3.2.1 Abort Scenario $Ab-S_1$

#### Update (and Update Latest) Operation

The protocol below shows how an update operation executes abort scenario  $Ab-S_1$ . Essentially, the update and update latest operations follow the same steps in implementing this algorithm.

---

#### *Update algorithm – Abort $Ab-S_1$*

---

1. Function **UPDATE-LATEST** (key, data-item)
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items) to DMS and TSM
4. DMS sends the UTID to the TSM to retrieve a start-time
5. **If** start-time is not issued
6.       Execute abort
7.       exit

#### Multi-key Operation

The algorithm for abort  $Ab-S_1$  followed by the multi-key operation is detailed below.

---

#### *Multi-Key algorithm – Abort $Ab-S_1$*

---

1. Function **Multi-Key** (key, data-item) 1..n
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items 1..n) to DMS and TSM
4. DMS sends the UTID to the TSM and gets a start-time
5. **If** start-time is not issued
6.       Execute abort
7.       exit



### 5.3.2.2 Abort Scenario Ab-S<sub>2</sub>

The Ab-S<sub>2</sub> is executed when any of the operations of a transaction fails.

---

#### *Multi-Key algorithm Abort Ab-S<sub>2</sub>*

---

1. Function **Multi-Key** (key, data-item) 1..n
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items 1..n) to DMS and TSM
4. DMS sends the UTID to the TSM and gets a start-time
5. DMS releases the data-item(1..n) to the TPE
6. TPE performs all the updates and sends back to DMS
7. DMS applies update and proceeds to TSM
8. If any operation fails, do:
  9. Initiate rollback on all successful operations
  10. Once rollback complete, execute abort
  11. Exit operation

### 5.3.2.3 Abort Scenario Ab-S<sub>3</sub>

#### **Update (and Update Latest) Operation**

The abort Ab-S<sub>3</sub> occurs when a transaction is unable to retrieve a commit time. As in abort Ab-S<sub>1</sub>, the algorithm for Update and Update-Latest follows the same procedure.

---

*Update Latest algorithm – Ab-S<sub>3</sub>*


---

1. Function **UPDATE-LATEST** (key, data-item)
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items) to DMS and TSM
4. DMS sends the UTID to the TSM to retrieve a start-time
5. **For each** data item  $\alpha$  in a read  $T_i$ , do:
  6.       If  $\alpha$  exists in any on-going transaction  $T_j$  in TSMs record, do:
    7.           Start *counter*
    8.           **while** *counter* < upper-bound time
      9.               wait for all  $T_j$  to commit
      10.              **if** *counter* >= upper-bound
        11.                 respond  $\alpha$  is unavailable
        12.                 exit
      13.              **else**
        14.                 DMS releases the data-item to the TPE
        15.                 TPE performs all the updates and sends back to DMS
        16.                 DMS applies update and proceeds to TSM
        17.                 For each data item  $\alpha$  in update  $T_i$ , do:
          18.                   **If**  $T_j$  *commit-time* between  $T_i$  *start-time* and  $T_{now}$  (where  $T_{now}$  is current-time)
            19.                       Initiate rollback operation
            20.                       Send abort  $T_j$
            21.                       End Transaction
          22.                   **Else**
            23.                       Issue  $T_i$  *commit-time*
            24.                       **if** ( $T_i$  *commit-time*)  $\rightarrow$  failed
              25.                           initiate rollback operation
              26.                           Send abort  $T_j$
              27.                           End Transaction

---

*Update algorithm – Ab-S<sub>3</sub>*


---

1. Function **UPDATE** (key, data-item)
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items) to DMS and TSM
4. DMS sends the UTID to the TSM and gets a start-time
5. DMS releases the data-item to the TPE
6. TPE performs all the updates and sends back to DMS
7. DMS applies update and proceeds to TSM
8. For each data item  $\alpha$  in update  $T_i$ , do:
  9. If  $\alpha$  exists in any on-going transaction  $T_j$  in TSMs record, do:
    10. If  $T_j$  commit-time between  $T_i$  start-time and  $T_{now}$  (where  $T_{now}$  is current-time)
      11. Initiate rollback operation
      12. Send abort  $T_j$
      13. End Transaction
    14. **Else**
    15. Issue  $T_i$  commit-time
    16. if ( $T_i$  commit-time)  $\rightarrow$  failed
      17. initiate rollback operation
      18. Send abort  $T_j$
      19. End Transaction

## Multi-key Operation

---

### *Multi-Key algorithm- Abort Ab-S<sub>3</sub>*

---

1. Function **Multi-Key** (key, data-item) 1..n
2. Generate UTID  $T_i$ , (at TPE)
3. Send (UTID, key items 1..n) to DMS and TSM
4. DMS sends the UTID to the TSM and gets a start-time
5. DMS releases the data-item(1..n) to the TPE
6. TPE performs all the updates and sends back to DMS
7. DMS applies update and proceeds to TSM
8. For each data item  $\alpha$  in multi-key  $T_i$ , do:
  9. If  $\alpha$  exists in any on-going transaction  $T_j$  in TSMs record, do:
    10. If  $T_j$  commit-time between  $T_i$  start-time and  $T_{now}$  (where  $T_{now}$  is current-time)
    11. Initiate rollback operation
    12. Send abort  $T_j$
    13. End Transaction
    14. **Else**
    15. Issue  $T_i$  commit-time
    16. if ( $T_i$  commit-time)  $\rightarrow$  failed
    17. initiate rollback operation
    18. Send abort  $T_j$
    19. End Transaction

### 5.3.3 Optimisation Decisions

In order to improve the performance of the system, certain design considerations have been taken in the implementation of the proposed system. For instance, to limit the number of aborts (or cascading aborts), the system introduces a variable known as *max-trax* which is the maximum number of allowed (concurrent) transactions that can take place on a key-item. Since the system makes use of snapshot isolation, it implies that operations are never blocked. This can lead to a high number of aborts when a particular data item is involved in many transactions. However, in such situations, the conflict would only be detected when requesting for a commit time. This can lead to a degrading performance

since transactions in this scenario have reached an advanced phase (*applied* phase). When too many transactions are aborting at this phase, it becomes a waste of processing resources. Once the number of on-going transactions involving a particular data item is equal to the max-trax value (which is configurable by application), the system would prevent transactions from accessing that data item till some of the transactions are completed.

Also, the decision to put TSM controllers in both DMS and TPE means that both components have access to the TSM. This decision is strategic as it means that the TPE can also interact with the TSM. This will improve the performance of the systems because when there is loss of information (such as a transaction timestamp) on the TPE, the TPE can retrieve that information directly from the TSM.

### **5.4 APPLICATION DOMAIN**

The application domain is used to test and evaluate the proposed system. The system designed implements a banking application known as the closed economy workload similar to the implementation in [121] which is evaluated in the next chapter.

Typically, a banking application will have multiple bank accounts and allow for banking transactions to take place across bank accounts. Operations that can take place on a bank account include checking accounts, cash deposits, cash transfers and deductions. Operations in the application must be atomic and consistent i.e. they must follow the ACID properties. These operations allows users to insert, update and delete records i.e., users can perform CRUD operations. In addition, users can perform multi-key operations such as transferring funds from one account to the other. The application data is stored persistently at the DMS layer which is the MongoDB database. Transactions take place at the TPE layer. So when a user makes a transfer operation (which is a multi-key transaction), the transaction takes the account details of both user and the amount to be transferred as arguments. The TPE receives the operation from the client and

requests for the user accounts details from the DMS following the protocol in section 4.6. After the transaction, the updates are sent back to the DMS. There are two collections (tables are known as collections in MongoDB) in the DMS which include: (i) accounts and (ii) transaction collections. Each document (records are referred to as documents in MongoDB) in the database has multiple attributes. The accounts collection contains the following attributes:

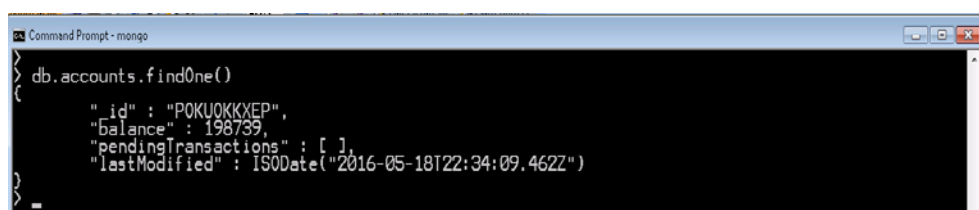
**\_id:** This is a unique identifier for each document in MongoDB. Every document in MongoDB must have an identifier attribute. When a new document is inserted in MongoDB, the value of the identifier must be specified. If it is not specified, MongoDB automatically assigns a value to the identifier (`_id`) attribute. In the proposed system, this attribute is used as a unique identifier for each account record. The unique identifier was generated using python codes.

**Balance:** The balance attribute is a user account attribute that stores the account balance for each record.

**PendingTransactions:** This attribute is an array type that stores the unique identifier for any ongoing transaction on that record i.e. it stores the transaction ID for ongoing transactions on that data item. Once a transaction is completed, its ID is pulled from the array of transaction IDs stored in this attribute. Since it is an array type, it can store multiple values. When there is no ongoing transaction, the value is an empty array (`[]`).

**lastModified:** As explained in [section 4.8](#), every record contains the *lastModified* attribute which is a timestamp data type. The attribute stores the commit timestamp issued by the TSM for the most recently committed transaction on that record.

Figure 5.8 below shows the record for a user account stored at the DMS.



```

Command Prompt - mongo
> db.accounts.findOne()
{
  "_id" : "POKUOKKXEP",
  "balance" : 198739,
  "pendingTransactions" : [ ],
  "lastModified" : ISODate("2016-05-18T22:34:09.462Z")
}

```

Figure 5.8: Account Details for an Account User

The second collection stored in the DMS is called the transactions collection. This collection stores information of all transactions that takes place in the system. The collection contains the following attribute.

**\_id:** This attribute, as mentioned earlier, is a unique attribute used to identify each document. For the transactions collection, the value of this attribute is not supplied by the user but automatically issued by MongoDB. This value is used to identify each transaction and represents the transaction ID for the application.

**State:** This attribute stores the state of the transaction which was explained in [section 4.4](#). It is updated anytime the transaction state changes. The value of this attribute is used to know the current state of any transaction. This includes aborted transactions.

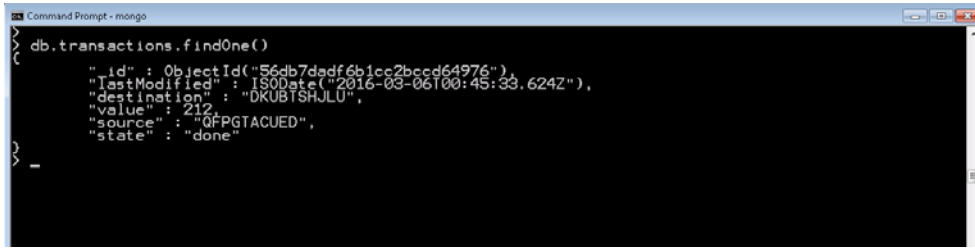
**Value:** The value attribute stores either the amount of money to be transferred from one account to another in a multi-key transaction. However, for an update operation, the attribute stores the new value supplied by the user which will be set as the new balance for an account in the accounts table.

**Source:** The Source attribute, stores the account ID of the user account from which the *value* would be deducted.

**Destination:** This attribute stores the account ID of the user account to which the *value* would be added.

**lastModified:** The *lastmodified* attribute in the transactions document stores the timestamp of the last time the transaction state was changed. This also allows the system to monitor the time in which each operation in a transaction was executed.

Figure 5.9 below shows the record for a transaction in the *done* state. The records are also stored at the DMS.



```

Command Prompt - mongo
> db.transactions.findOne()
{
  "_id" : ObjectId("56db7dadf6b1cc2bccd64976"),
  "lastModified" : ISODate("2016-03-06T00:45:33.624Z"),
  "destination" : "DKUBTSHJLU",
  "value" : 212,
  "source" : "QFPGTACUED",
  "state" : "done"
}
-

```

Figure 5.9: Transaction Records

The operations were implemented in the following methods.

**ReadOp()** – The ReadOp method takes a user account ID as an argument, and returns the account details record for that account ID.

**Read-LatestOp()** – This method also takes a user account ID as an argument and returns the account details for that account if there is no ongoing transaction.

**UpdateOp()** – The UpdateOp methods takes two arguments which includes the account identifier and the new balance for that account. It returns a success message or a failure message depending on the outcome of the transaction.

**Update-LatestOp()** – The Update-LatestOp methods also takes two arguments which includes the account identifier and the new balance for that account. It returns a success message or a failure message depending on the outcome of the transaction. If there is an ongoing transaction, it waits for a user-specified time before.

**MultiKkeyTransactionOp()** – The multikeyTransactionOp is a method that invokes transaction that involves multiple key items and operations. It involves the transfer of money from one account to the other and also maintains ACID properties of a NST transaction (see [section 4.1](#)).

In order to maintain consistency in the operation of multi-key transactions, the constraint holds that the total money in the closed economy is an invariant i.e. when money is transferred from one account to the other, the transaction must be atomic, leaving the database in a consistent state.



### 5.5 SUMMARY

This chapter provided an analysis of the proposed system. The design objectives which guided the approach taken in this system were clearly explained. The proposed approach aimed to provide consistency and to maintain the ACID properties of transaction. The system design put into perspective the fact that cloud database systems need high availability and efficiency. Therefore, the implementation ensured that availability was not sacrificed while trying to achieve consistency. The implementation makes use of well-known industry tools and technologies such as MongoDB and SQLite. The programming language used in implementation, Python 2.7, includes libraries that can interact with MongoDB, SQLite and a host of other cloud and relational databases. This provided a seamless interaction between the components of the system. The cloud layer was managed by Ubuntu Juju which is a cloud hosting platform that allows users to easily manage nodes in a cluster.

The chapter also explained the different types of operation that is supported by the prototype system. Three of these operations which are novel to cloud systems include the read-latest operation, update-latest operation and the multi-key operation. The algorithm followed in the implementation of these operations are outlined and explained. Operations in any database can abort and the system must have a way to handle such scenarios to preserve consistency. This chapter explained the different scenarios that can cause an operation to abort. The chapter then explains the procedure followed by the proposed system to handle these scenarios for each of the operations.

Finally, this chapter explained the application domain which was used to implement the prototype system. The domain simulates a banking application which allows users to perform CRUD operations. The different components and attributes used in the implementation were explained. Since the system uses MongoDB to implement the DMS, the application stores its data persistently in MongoDB. The different data entities used in the implementation were explained.

The next chapter explains the various experimentation and metrics used to evaluate the system. The chapter also analyses the results of the experimentation.



# CHAPTER 6

## EXPERIMENTAL EVALUATION

This chapter evaluates the proposed system using workloads derived from well-known standard cloud benchmarks which include the YCSB (Yahoo! Cloud Service Benchmark) and the YCSB+T (Yahoo! Cloud Service Benchmark and Transaction). The results of the evaluation are documented in this chapter. The proposed system is also evaluated in comparison to existing systems.

The evaluation of the proposed system is carried out by taking into account failure free environment. That is, failures such as system, network communication, etc., are not considered. This is in line with most of the existing approaches which consider failure free environment in the evaluation of transaction processing both in the classical databases as well as cloud databases.

Section 6.1 explains the benchmark and the workloads used to evaluate the proposed system. Section 6.2 explains the various experiments carried out and presents the results of the experiment. Section 6.3 analyses the proposed system in comparison with other similar systems.

### 6.1 EVALUATION BENCHMARKS AND WORKLOADS

The evaluation of the prototype system was carried out using a combination of the two widely used benchmarks, YCSB [122] and YCSB+T [123] cloud benchmarks. The YCSB benchmark is recognised as a standard benchmark used to evaluate cloud database while the YCSB+T benchmark was developed as an extension to the YCSB benchmark. The next section explains the two benchmarks as well as the workloads used in the evaluation

### 6.1.1 YCSB and YCSB+T Benchmark

The workloads developed for evaluating the proposed system, (explained in the next section) were adopted from a combination of the YCSB and the YCSB+T benchmarks. The YCSB is the most widely accepted cloud data benchmark and was developed to evaluate performance of cloud data serving (distributed) systems. The YCSB benchmark evaluates performance by focussing on the latency of requests to the cloud database. The benchmark also aims to measure the trade-off between latency and throughput in cloud systems. As such, some of the workloads implemented in this evaluation were adopted from this benchmark. However, the benchmark was not designed to evaluate transactions and consistency, since most cloud databases do not provide transaction support. The YCSB+T benchmark was therefore designed as an extension of the YCSB benchmark to evaluate cloud databases that offer support for transactions. The YCSB+T take into consideration, the need to preserve ACID properties during the execution of operations in a transaction. Therefore, the YCSB+T benchmark is designed to detect consistency anomalies introduced during the execution of the transactions.

Therefore, the approach used in this evaluation is a combination of both benchmarks. The next section explains the workloads.

### 6.1.2 Workloads for Experiments

The proposed system has five (5) types of operations which were explained in [section 5.3.1](#). The operations evaluated include the following:

**Read:** The read operation takes a data key identifier as an argument and returns the details for that key. Read operations are never blocked and will always return a result.

**Read-Latest:** This also takes a data key identifier as an argument and returns the details for that key if there is no running transaction on that key item. If there is, it

## EXPERIMENTAL EVALUATION

waits for a specified amount of time and returns a fail if the specified time is elapsed and the transaction is still running.

**Update:** This takes a data key identifier and the detail to be updated as arguments. It retrieves the data and performs an update on the data.

**Update-Latest:** Update-Latest also performs an update on a data item if there are no running transactions involving that data item.

**Multi-key:** The multi-key transaction involves more than one data items and operations. The execution of a multi-key transaction follows the constraints of a NST transaction defined in [section 4.1](#) and preserves the ACID properties of transactions.

For read-latest and update-latest operations, an upper-limit value is arbitrarily chosen by a user, explained in ([section 5.3.1.2](#) and [section 5.3.1.5](#)). The upper limit value determines how long an each of these two operations must wait for an ongoing transaction before it times-out and decides to abort. This is similar to timeout mechanisms applied in existing transactional protocols. For instance, in the [2PC protocol](#), the coordinator and participants have to wait for message from each other. To prevent unnecessary delays, the system protocol times out after a certain amount of time. The termination and restart protocol (explained in [section 2.4.1](#)) determines how the system behaves when such timeouts occurs.

In choosing upper limit values for the Read-Latest and Update-Latest operations, the following factors were put into consideration.

- (1) A very high upper limit time means transactions may have a higher latency which is not ideal for applications
- (2) A very high upper limit time would mean that there would hardly be any abort due to conflicts since the transactions would wait longer for ongoing transactions to complete. This will defeat the purpose of evaluating the system as it would be difficult to evaluate how the system reacts to conflicting operations.
- (3) A very low upper limit time will make it difficult to see the difference between an ordinary read (or update) transaction and a read-latest (or update-latest) transaction.

## EXPERIMENTAL EVALUATION

Putting these into consideration, the upper-limit time was set to two (2) seconds and three (3) seconds for read-latest and update latest operations respectively. The values were different for the two operations because it is expected that an update operation would normally take longer to process than a read operation. There is no generally accepted standard for setting transaction timeouts. It may vary from one protocol (or application) to another.

As stated earlier, the workloads adopted for this thesis were a combination of the workloads used in the YCSB and YCSB+T benchmarks. Workload A, B and C were adopted from the YCSB benchmark. Workload E and G were adopted from YCSB+T benchmark and was used in this thesis to evaluate performance difference between strictly read operations and strictly write operations. Workloads D and F were developed for the purpose of this research as both workloads are made up of operations that novel to cloud database systems. The YCSB and YCSB+T benchmarks do not contain these operations. The application domain used to evaluate this system is closely similar to the YCSB+T benchmark which simulates a banking application. Table 7.1 shows the workloads mixture used to evaluate the system.

**Table 6.1: Workloads for Evaluation**

WORK LOAD	TYPE	OPERATIONS
A	Read only	Read 100%
B	Read-heavy	Read 90%, Update 10%
C	Update heavy	Read 50%, Update 50%
D	Read Latest heavy	Read latest 90%, Update 10%
E	Update Only	Update 100%
F	Multi-key heavy	Multi-key 50 % and Update-latest 50%
G	Multi-key transaction	Multi key transactions 100%

The explanation of the operations in each workload is as follows.

**Workload A:** This workload contains only read operations. Therefore all the client operations generated by this workload are limited to read operation i.e., 100% read operation.

## EXPERIMENTAL EVALUATION

**Workload B:** This workload contains a mixture of read and update operations. This would help to assess if the system is optimized for read or for write operations. 90% of the operations generated are read operations and the remaining 10% are update operations. Therefore it is called Read-heavy.

**Workload C:** This workload also contains a mixture of read and writes operations. However, the ratio of read to write operation is 1:1. Therefore, 50% of the operations are read operations while update operations also represent 50%.

**Workload D:** Workload D aimed to assess the impact of the read-latest operation on performance and latency. As stated earlier, the upper-limit time for read-latest operation was set to two (2) seconds. Therefore, this workload was made up of 90% read-latest and 10% update operations.

**Workload E:** This workload is made up of only update operations. The Workload is compared with workload A to compare read versus write performance. The workload is therefore 100% update operations.

**Workload F:** This workload is a mixture of multi-key operations and update latest operations. Both operations a write operations and is designed to see the performance of the system under heavy writes.

**Workload G:** Workload G is 100% multi-key operations. This workload is carried out to assess the impact of maintaining ACID properties on the system.

Each of these workloads was run on various experiments based on certain metrics which are explained in the next section. These metrics are used to evaluate the system.

The data to be used for the evaluation was loaded into the data management store (DMS). For workloads A, B, C, D and E, one thousand (1000) unique data items were loaded into the system while for workloads F and G, two thousand (2000) unique data key items were loaded into the system. Workloads F and G both contain two thousand data items because these workloads include multi-key operations which involves multiple key items. For a cloud system, one thousand (or two thousand) key items are relatively low. However, this number was chosen because the lower the number of data items involved in transactions, the higher

the chances of a conflict. This would allow us to evaluate the consistency of the system. For each of the workloads, one thousand requests per client are generated. The requests are randomly generated to perform operations on the pool of data loaded into the system. The requests are generated for 1, 2, 4, 8, 16 and 32 clients to check the performances of the system under load (each client generates one thousand requests). Therefore, at 32 clients, with each client generating one thousand requests, total workload generated will be thirty two thousand (32,000) requests. Table 7.2 shows the total number of operations generated by the clients. In the YCSB+T benchmark, performance of transactions were measured from one (1) to sixteen 16 clients. However, for this thesis, each experiment is extended to thirty-two (32) clients. This is to add extra load to the system.

**Table 6.2: Number of Clients and Operations Executed**

<b>Number of Clients</b>	<b>Operations Generated per Clients</b>	<b>Total Number of Operations Executed</b>
<b>1</b>	1,000	1,000
<b>2</b>	1,000	2,000
<b>4</b>	1,000	4,000
<b>8</b>	1,000	8,000
<b>16</b>	1,000	16,000
<b>32</b>	1,000	32,000

## **6.2 EXPERIMENTS AND RESULTS**

Based on the above workloads, various experiments have been conducted in order to evaluate the proposed approach in relation to the objectives of this research set out in Chapter 1.

Different sets of experiments are conducted by taking into account different factors. These include:



### **Set 1: Number of transactions per second:**

This experiment measures the throughput (i.e. number of completed transactions per second) that the proposed system can handle for each of the workloads. All the workloads are run, and the number of completed transaction per second is recorded.

### **Set 2: Latency of each operation**

This set of experiments measure the latency each operation in each of the workloads while varying the number of client threads. The latency of each operation measures the average time it takes to complete an operation.

### **Set 3: Percentage of total completed transactions**

As stated earlier, the experiments assume a failure free environment. However, the evaluation also considered that there are situations which would lead to transaction aborts. This set of experiment measures the percentage of operations in each workload that ends with a successful commit.

### **Set 4: Number of failed transactions**

This experiment measures the total number of transactions that were aborted for each of the workloads.

### **Set 5: Distribution of failed transactions**

This set of experiments measures the distribution of aborted transactions. Recall from [section 5.3.2](#), that there are three scenarios which can lead to aborts. This experiment measures the percentage distribution of each of these scenarios that caused the aborts.

### **Set 6: Overall latency of each workload**

These experiment measures the total time it takes for each of the workloads to complete executions. Table 6.2 shows the total number of operations that are executed for each operation.

### **Set 7: Consistency Overhead**

The consistency overhead measures the performance cost (measured by latency of workloads) of implementing consistency in the system. The experiment compares the latency of an update operation performed directly on the DMS with the latency of update operations performed via the TPE. Operations performed via the TPE implements snapshot isolation and provides stronger consistency guarantees.

### **Set 8: Anomaly scores**

The consistency of the system is evaluated by measuring the correctness multi-key transactions (workload G). The correctness of transactions will be evaluated on varying number of clients (using 1, 2, 4, 8, 16 and 32, client threads). This will involve a consistency check on the records in the DMS to determine if there is any anomaly. An anomaly is scenario in which the system deviates from the expected behaviour which could affect the consistency of the system. For instance, if a transaction fails to complete a roll back before it aborts, this represents an anomaly and leaves the database in an inconsistent state. The calculation for the anomaly score is adopted from the YCSB+T benchmark. An anomaly score is defined in [119] as the number of anomalies that is introduced into the system during the run of the workloads. This evaluation is carried out after each run of workload G. The expectation is that a consistent system should have an anomaly score of zero.

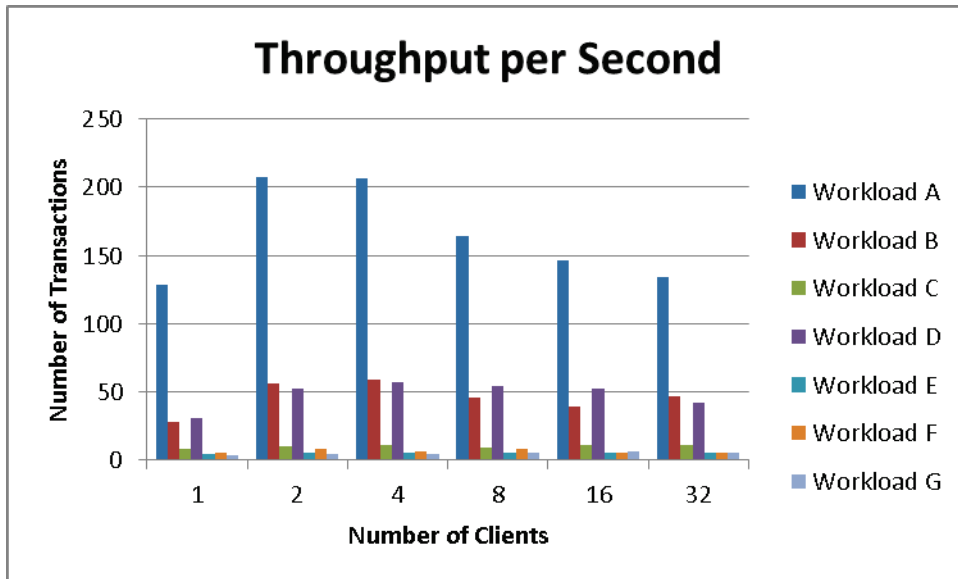
#### **6.2.1 Experiment – Set 1**

This experiment evaluates the ‘number of transactions per second’. The following table (Table 6.3) shows the throughput (i.e. number of transactions per second) that the prototype system can handle for each of the workloads.

**Table 6.3: Total Number of Completed Transactions per Second**

No of Clients	Workload A	Workload B	Workload C	Workload D	Workload E	Workload F	Workload G
	(Read 100%)	(90% read, 10% Update)	Update heavy	(90% read-latest, Update 10%)	(Update 100%)	(50% multi-key,	Multi key (100%)
1	128	28	8	31	4	5	3
2	207	56	10	52	5	8	4
4	206	59	11	57	5	6	4
8	164	46	9	54	5	8	5
16	146	39	11	52	5	5	6
32	134	47	11	42	5	5	5

Note that the YCSB+T and YCSB benchmarks do not specify the size of data use for transactions. However, the average size of each key item (or document as in MongoDB) used in each operation is 112 bytes. Each multi-key transaction contains two key items (and 4 read and write interleaved operations). The data in the table above is expressed as a graph below to show the relationships between the workloads.



**Figure 6.1: Transactions per Second**

Figure 6.1 above shows the number of operations (and transactions) per second the prototype system can handle. The figure also shows that the system is highly optimised for reads. Other operations such as the Update, Read-latest, Update-Latest and Multi-key transactions are relatively slower because they provide a stronger consistency guarantee. In order to achieve that, they incur extra message

## EXPERIMENTAL EVALUATION

communication and processing delays. Each of these operations must contact the TSM for information on previous transactions and wait for the response of the TSM before they can progress. This introduces some form of latency to the system. However, this is a trade-off that must be made for the system to guarantee consistency. For example, a simple update operation without any consistency check will take an average of 0.009s/ operation while adding consistency check will increase the average time for a transaction to 0.2s. This is because adding consistency will mean every transaction must make at least 2 trips to the TSM. The additional latency,  $L_u$ , incurred by an update (update and update latest) operation with consistency is explained with the equation below.

$$L_u = [4 T_{tsm} + T_{st} + T_{ct}] \quad (6.1)$$

Where  $T_{tsm}$  is the time taken for a network message trip between the DMS and the TSM (i.e. Each round trip to receive a timestamp from the TSM represents 2 network messages journeys to and fro),  $T_{st}$  is the time taken for the TSM to process and issue a start time and  $T_{ct}$  is the time taken for the TSM to process and issue a commit time. Ideally, it is expected that  $T_{ct}$  would be greater than  $T_{st}$  because before  $T_{ct}$  is issued, the TSM must check all on-going operations as well as operations that started after the current operation start time to ensure that there are no conflicting operations. For read operations, the latency is lower because a read operation does not need any form of contact with the TSM.

The equation is slightly more complex with multi-key transaction because a multi-key transaction involves multiple data items. It involves a new variable  $N_d$  which is the number of data items involved in a transaction. Therefore, for a multi-key transaction, equation (1) above is modified as follows:

$$L_u = [(N_d * 4 T_{tsm}) + T_{st} + T_{ct}] \quad (6.2)$$

The system is able to handle roughly 207 read operations per second at its peak. For write operations (update, update-latest and multi-key), the number of transactions in a second is reduced. For instance, the result shows that the throughput of Workload G is about six transactions per second. This is expected as a write operation would definitely incur higher latency than read operations due to the forced-writes or logs involved. Also, the new operations read-latest and

## EXPERIMENTAL EVALUATION

update latest have lower level of throughput than read update operations consecutively. Again this is an expected behaviour. These two operations give users the power to determine the level of consistency that they fill would be suitable for their application. Thus, a higher value for the upper-limit would lead to a stronger the consistency guarantee. However as stated earlier on, this would be a trade-off on latency as the transaction can be slower depending on the number of ongoing transactions on that data item. This approach to calculating the network latencies has been used in previous research. For instance in [16], the latency various protocols were calculated using this approach. The performance of 2PC is measured using the number of messages exchanged, forced-write operation, etc.

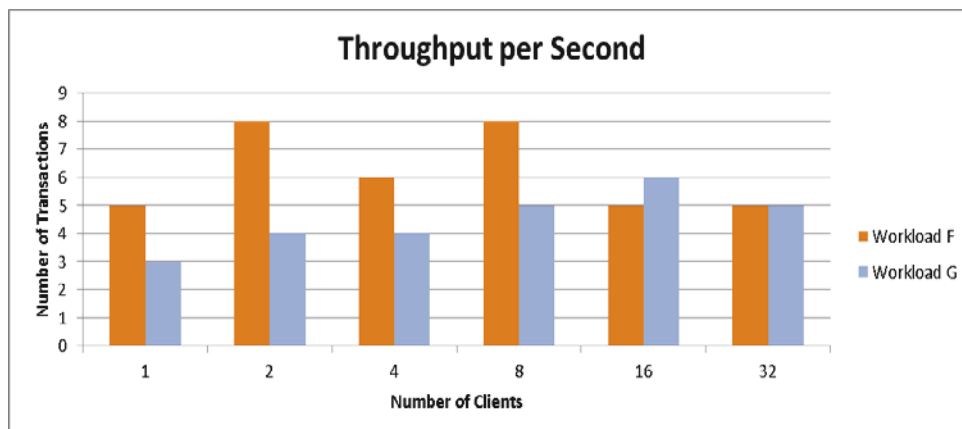


Figure 6.2: Workload F vs. Workload G

### 6.2.2 Experiment – Set 2

This experiment measures the average time taken to complete an operation for each of the workloads.

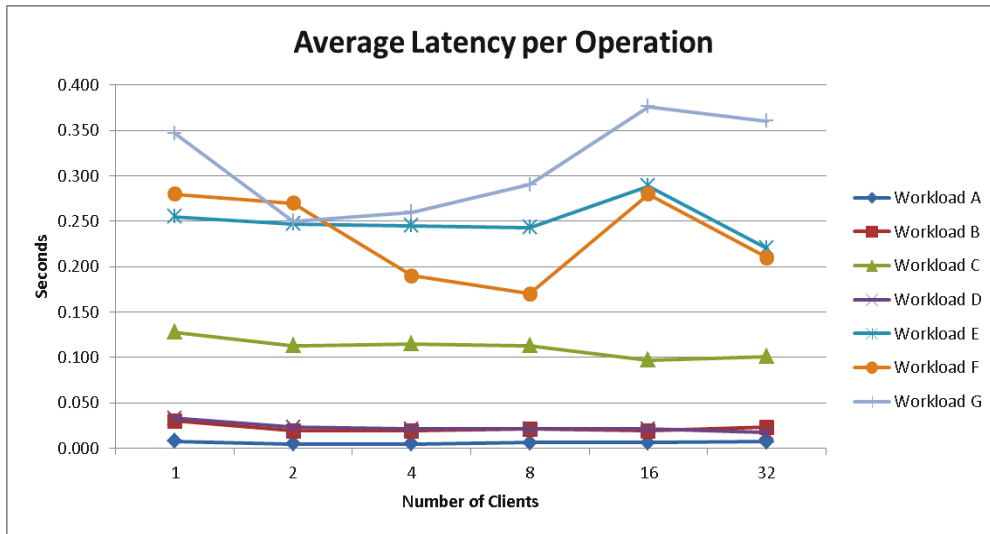


Figure 6.3: Average latency per transaction

The Figure 6.3 above shows that a read transaction will take an average of 0.006 seconds, an update transaction will take an average of 0.2 seconds and a multi-key transaction will take an average of 0.3 seconds to complete. This shows that the system is able to provide a lower latency than some existing systems. For instance, Megastore [51] has a latency of about 0.4 seconds for a write operation and up to tens of milliseconds for read operations which is relatively higher than the proposed system.

### 6.2.3 Experiment – Set 3:

This experiment evaluates the “Percentage of Total Completed Transactions”. As stated earlier, for each of the workloads, one thousand requests were generated per client. The graph in Figure 6.4 shows the percentage of requests that were completed for workloads A - F. A completed request means operations and transactions that were committed i.e. read operations that returned a result and successful write operations (and transactions). Recall from the definitions and constraints of a *NST* in section 4.1 that a *NST* is of type seq (*Begin* | *OP<sub>i</sub>* | *Cmt* | *Abt*) with the condition that either *Cmt* (commit) or abort (*Abt*) occurs only once within the sequence. The formula below shows the calculation for the percentage transaction completion rate  $PT_{cr}$ .

$$PT_{cr} = [(N_{ct} / N_r) * 100] \text{ (eq. 3)}$$

## EXPERIMENTAL EVALUATION

Where  $N_{ct}$  is the total number of completed transaction per workload and  $N_r$  is the total number of client requests issued.

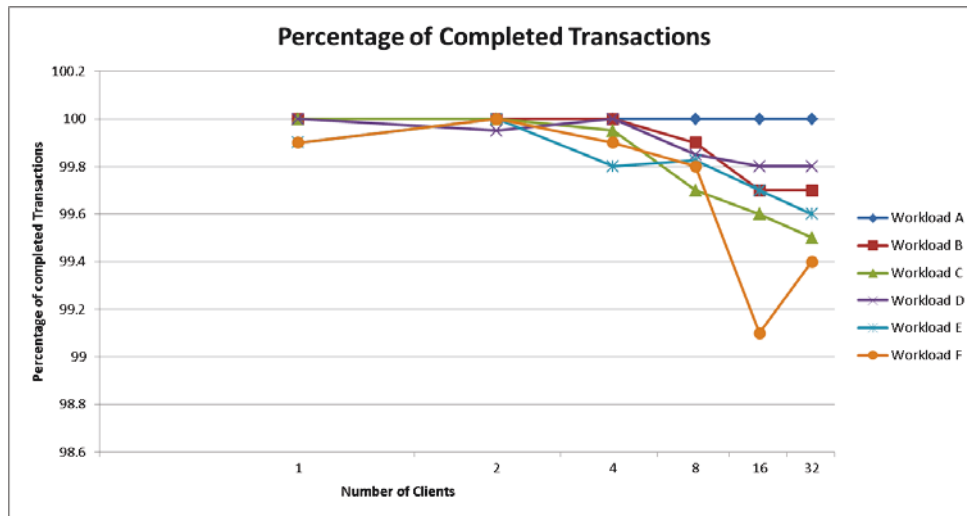


Figure 6.4: Percentage of completed transactions

For read operations (workload A), all requests were completed without any aborts. This is in line with the operations in snapshot isolation as explained in section 5.1.1. However, for other workloads, there were a few aborted operations due to failure of the TSM to issue start-time or commit-time. However, most of the transactions were able to maintain above 99% of completion with the exception of workload G which is in Figure 6.5 below.

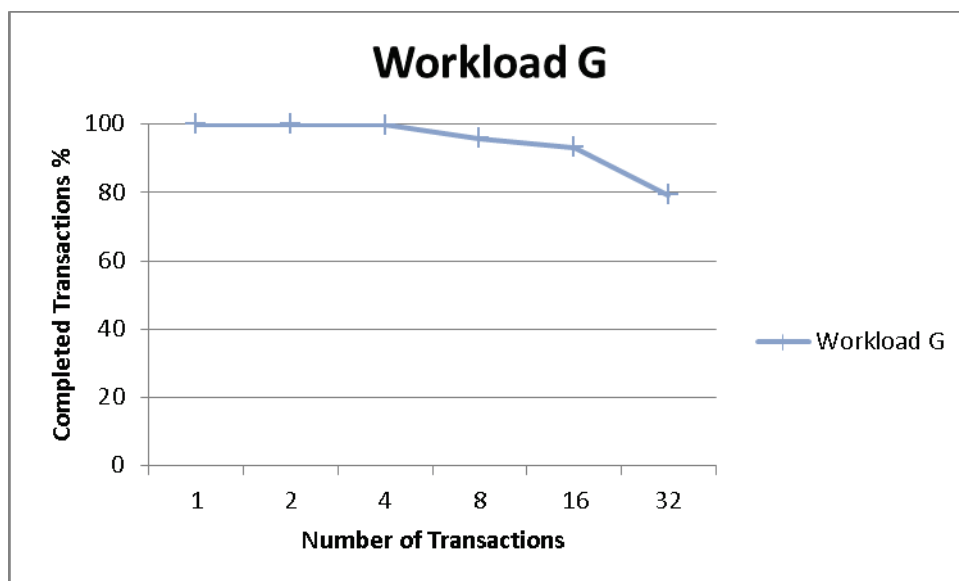


Figure 6.5: Percentage throughput of Multi-key transactions

Workload G had relatively lower percentage throughput ( $PT_{cr}$ ) when compared to other workloads. Again, this is expected as the multi-key operation is a transaction involving more than one data items and multiple operations. Also recall from definition one in chapter four, a *NST* is a sequence of operations which are executed in a way such that all of them are successfully completed or none at all. This is to preserve the atomicity property of transactions. As such, it is expected that a multi-key operation will have a lower throughput than other operations. However, even with thirty-two clients (32,000 operations) the system is able to process to completion over 80% of transactions.

#### 6.2.4 Experiment – Set 4

The figure below shows the total number of aborted transactions for workloads A to F.

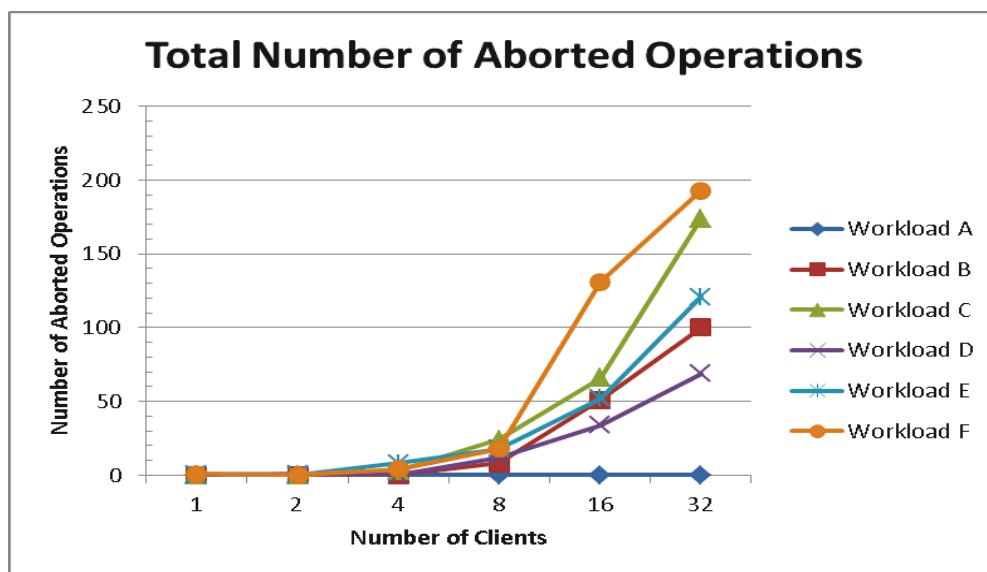


Figure 6.6: Total Number of Aborted Transactions

The figure 6.6 shows that there were no aborted operations in workloads A. All other workloads show relatively low numbers of aborted transactions. The Figure 6.7 below shows that multi-key transactions have a relatively higher number of aborted transactions when compared to other workloads. Figure 6.7 also shows a



comparison between the number of aborted transactions in workload A (read only operations) and workload G (multi-key operations).

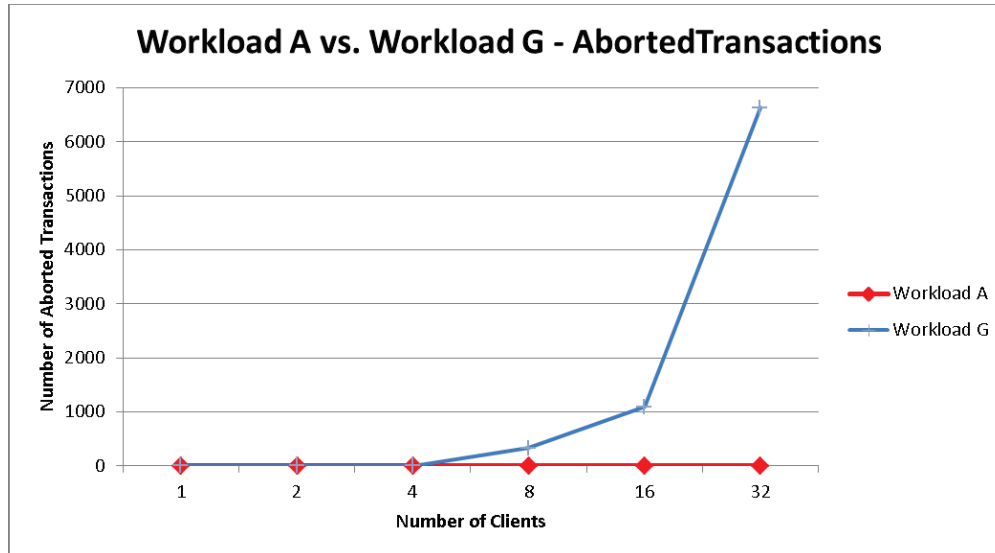


Figure 6.7: Workload A vs Workload G (Aborted Transaction)

Generally, most of the aborts were as a result of the inability of the TSM to meet up with the speed of incoming request when issuing transaction start-times. When an operation is unable to receive a start timestamp, it will lead to an abort type  $Ab-S_1$  explained in [section 4.7](#). This caused a very high percentage of the aborts as most of the aborted transactions occurred when the transaction was in the *initial* state. Also, because the multi-key transactions involved multiple data items, each of these data items could be involved in other transactions which would lead to a higher number of aborts in workload G. Future work will implement queues to control the rate at which the client requests arrive at the TSM. This can improve the rate of transaction completion but will however incur a higher latency.

The number of  $Ab-S_2$  and  $Ab-S_3$  aborts was very low. The main cause of an  $Ab-S_3$  abort is when one part of a transaction should fail leading to a total abort of the operation. On the other hand an  $A_3$  abort occurs when a transaction is not issued with a commit timestamp mainly as a result of a conflicting transaction. The rate of data conflict occurrence (transactions aborted at "*applied*" state) was very low as such; transactions that aborted at the point of retrieving transaction commit-

## EXPERIMENTAL EVALUATION

time were very rare. This is because for this type of failure to occur, another transaction must have its commit-time in the interval between the start-time and commit time of an ongoing transaction. For instance, for an abort  $Ab-S_3$  to occur on a transaction  $T_i$ , then a transaction  $T_j$  must have its commit time in the interval  $S$  (see figure below) and must contain one or more data items in transaction  $T_i$ .

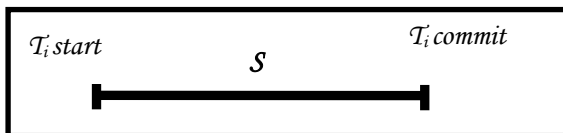


Figure 6.8: Period between transaction start time and commit time

Where  $S$  is interval/period where the transaction  $T_i$  takes place,  $T_i$  start is transaction  $i$  start time and  $T_i$  commit is transaction  $i$  commit time.

As Figure 6.8 above shows, for a multi-key transaction which has the highest latency,  $S = 0.3s$ . Since our request was pulled randomly from a set of one thousand data, the probability of  $Ab-S_3$  happening was very low. The next reading shows the distribution of the aborted transactions.

### 6.2.5 Experiment – Set 5

This set of experiments measures the distribution of aborted transactions. As mentioned earlier, the three types of aborts that can occur include  $Ab-S_1$ ,  $Ab-S_2$  and  $Ab-S_3$ . Abort  $Ab-S_1$  has the highest frequency. As explained, an abort  $Ab-S_1$  occurs when a transaction (or operation) does not get a start timestamp. At this stage, the transaction is in an *initial* phase. As soon as a start time is issued, a transaction enters into a *pending* state where write operations take place. If any part of the write operation fails, then a rollback would begin, followed by an abort. This is abort  $Ab-S_2$ . If all operations are successful, the transaction moves into an *applied* state and proceeds to retrieve a commit time from the TSM. Failure of a transaction to retrieve a commit time will lead to abort type  $Ab-S_3$ . Most of the aborts however are caused by the TSM. As such, for workloads B and C, result showed that most of the aborts occurred when the transaction was still at *initial*

## EXPERIMENTAL EVALUATION

stage. This was because the transactions were unable to receive a start-time because the TSM could not meet up with the speed of the request. The TSM therefore represents a bottle neck and a single point of failure for the system. Future work will look at ways in which the TSM can be further optimised, in order to ensure proper synchronization across replicas of the TSM. The table 6.4 below shows the state at which transactions were when they were aborted.

State	Workload A	Workload B	Workload C	Workload D	Workload E	Workload F	Workload G
Initial	0	99.4 %	99.6%	98.3%	95.5%	93.37%	99.7%
Pending	0	0.6%	0.4%	1.7%	0.5%	6.34%	0.3%
Applied	0	0	0	0	0	0.29%	0
	0	100	100%	100%	100%	100%	100%

**Table 6.4: Percentage distribution of Transaction state when abort took place**

The diagram above shows that a very high percentage of the total aborted transactions in each workload took place when the transactions were still in their *initial* state. Though this affects the throughput but it does not affect the consistency of data. That is, in the *initial* state, transactions do not manipulate (update) data and thus there is no risk of data inconsistency.

### 6.2.6 Experiment – Set 6

This reading evaluates the total time taken for each of the workloads to complete one thousand requests per client. The latency measured includes a combination of both committed and aborted requests. The Figure 6.9 below displays the results for latency of total request for each of the workloads. The results show that the system is able to process clients request at relatively low latencies. The system can complete thirty-two thousand (32,000) read transactions in about 236 seconds. For a single client, the system can process one thousand read operations in just less than 8 seconds.

## EXPERIMENTAL EVALUATION

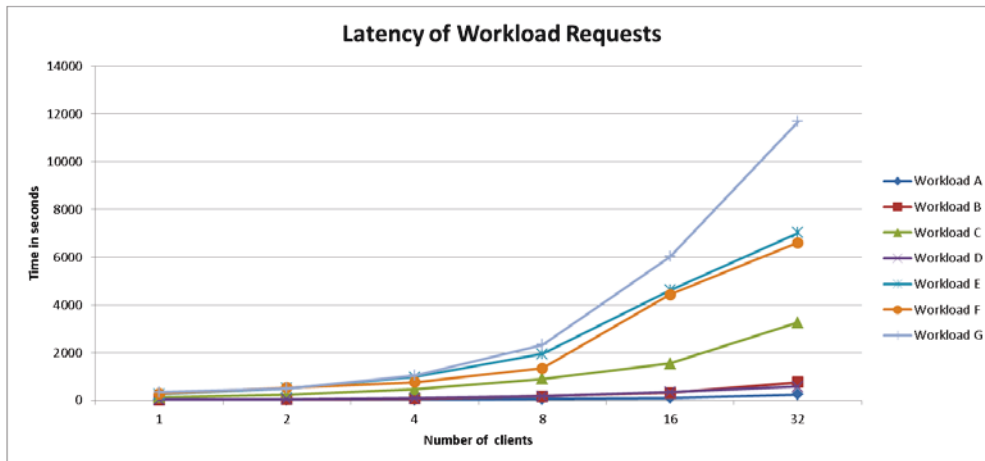


Figure 6.9: Latency of Workload Requests

### 6.2.7 Experiment – Set 7

This experiment measures the performance cost of implementing consistency. The Figure 6.10 below compares the latency of a simple update operation on the DMS (with no consistency) with the latency of our update operation which is able to guarantee stronger consistency by implementing snapshot isolation (using the TSM). This result shows the overhead (in terms of latency cost) involved in guaranteeing consistency. The Figure 6.10 shows that in order to maintain consistency, the system incurs a higher level of latency due to the extra processing and messaging time. See equation (6.1) of [section 6.2.1](#). However, this is a trade-off between latency/performance and guaranteeing stronger consistency.

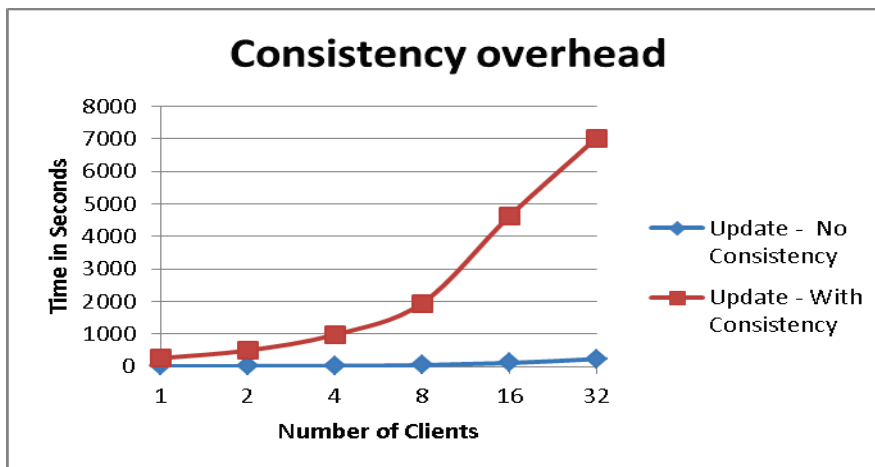


Figure 6.10: Transactional Overhead of Update Operations

### 6.2.8 Experiment – Set 8

An anomaly occurs when the system deviates from its expected behaviour leading to errors and violating consistency guarantee provided by the system. As mentioned earlier, the method for calculation for anomaly was adopted from the YCSB+T benchmark and was also adopted in [121]. An example of anomaly in this case is if a transaction aborts mid-way and the rollback does not occur. Another anomaly that can occur is the failure of the TSM to detect conflicts in transactions. As mentioned [section 5.4](#), the implementation domain used to evaluate the prototype system known as the closed-economy workload. This is also adopted from the YCSB+T benchmark that models a banking environment. The formula for calculating the anomaly score is given below.

$$\gamma = \frac{|S_{initial} - S_{final}|}{n}$$

Where  $\gamma$  = simple anomaly score,  $S_{initial}$  = initial sum of all the accounts,  $S_{final}$  = final sum of all the accounts and  $n$  = number of operations executed. The table below shows the anomaly score for multi-key workloads.

Number of Clients	Number of operations	Initial sum	Final sum	Anomaly score
1	1000	400000000	400000000	0
2	2000	400000000	400000000	0
4	4000	400000000	400000000	0
8	8000	400000000	400000000	0
16	16000	400000000	400000000	0
32	32000	400000000	400000000	0

Table 6.5: Anomaly Score

The formula is a perfect fit for evaluating the consistency of multi-key transactions as a multi-key transaction must leave the database in a consistent state. In the banking application domain, the total amount of money in the economy is an invariant. Therefore, during transfer of funds from one account to the other (which is essentially a multi-key transaction), the amount deducted from one

account must be added to another account to complete the transaction atomically and leave the database in a consistent state. As such, the sum of all accounts must remain an invariant irrespective of the number of operations (or transfers in this case) executed. Therefore, any difference between the initial sum of accounts and the final sum of the accounts after the execution of the workload would imply that the database is in an inconsistent state. The anomaly score is however calculated in relation to the number of operations because the higher the number of request, the higher the possibility of inconsistencies. Table 6.5 shows that our system maintains an anomaly score of zero irrespective of the number of clients or number of operations. This shows that the system has a strong consistency guarantee with no errors. The price of this however is a slightly higher latency for transactions. A comparison of the anomaly scores derived from this evaluation with anomaly scores for ReTSO in [121] shows that this system does better than ReTSO with respect to consistency guarantees. The experiments were designed to identify the anomalies present in the execution of operations. The results show that 11% of the total number of failed transactions was due to consistency anomalies in ReTSO.

### **6.3 ANALYSIS OF THE PROPOSED SYSTEM AND EXISTING APPROACHES**

In this thesis, the evaluation of the proposed system is carried out using two standard benchmarks, YCSB and YCSB+T, rather than comparing it to a single existing approach. The evaluation therefore provided an in-depth analysis of the various features of the proposed approach and has determined that the proposed approach has met the objectives set for this research.

Further, the decision of not restricting the evaluation of the proposed approach to a single existing approach is that there are significant differences in the transaction models, protocols, design and implementation of the proposed system and those of existing systems.

The architecture of the system implemented in this thesis is similar to the architecture in Deuteronomy [102] and Megastore [51] which follows the

middleware approach explained in [section 3.3.2](#). The rationale behind the choice of this middleware architecture is that it allows our system to separate transactional functionalities from storage functionalities. However, the method of implementing transactional semantics deviates from the implementation in Deuteronomy [102] [113]. Deuteronomy makes use of data locking which adds extra overheads and reduces concurrency.

The proposed system implements transactional semantics in a way similar to [104] using an optimistic concurrency control technique which allows for higher level of concurrency. Compared to the approach in [104], the critical and novel aspect of the implementation in this thesis is the commit ("*lastmodified*") timestamp parameter which is associated with every data item – a commit timestamp that allows the proposed system to identify the replica with the latest version of any key item. Since the prototype system does not follow the one-copy serializable transaction model in [111], the "*lastmodified*" timestamp parameter becomes an important aspect of our system for maintaining consistency. This removes the extra effort required to maintain one-copy serializable transactions and does not jeopardize the consistency of the database. Maintaining one-copy serialization can affect availability negatively since replicas are not allowed to be out of date. This can lead to rejecting transactions when a replica is failed.

Moreover, the proposed system incorporates a Timestamp Manager, similar to ReTSO [104] which allows the system to manage transactions across the system. ReTSO also makes use of snapshot isolation but has a different architecture and different set of operations. The proposed system design has taken advantage of this centralized time manager to introduce new types of operations called update-latest and read-latest. A brief comparison with ReTSO shows that ReTSO has a higher throughput than the proposed system [104]. However, ReTSO stops short of providing multi-key transactions and cannot guarantee consistency at the level of our system.

Megastore [51], G-store [100] and [106] all implement transactions using the middleware architecture as in our system, however, transactions can only occur among data in a subset of the total data in the system (typically among data that share a common boundary). The proposed system can implement transaction

across all the data items in the system and does not limit transactional capabilities to a subset of data (or entity groups). As stated earlier, experimental results also shows that the proposed system has a lower latency per operation than Megastore [51]. The results show that Megastore has a write latency of about 0.4 seconds and read latency of up to tens of milliseconds. The proposed system has a write latency of between 0.2 and 0.3 seconds and a read latency of 0.006 seconds.

### 6.4 SUMMARY

The aim of this research was to implement transactions in a NoSQL cloud database. In line with the design objectives highlighted in section 5.1, the prototype implementation aimed at achieving high availability, high concurrency control without sacrificing consistency. This evaluation has been able to prove that these objectives were met. The above results show that this system is able to provide multi-key transaction support on a cloud database.

In summary, the system is able to reliably manage transactions with ACID level consistency. When a failure occurs or a conflict happens in a transaction, the system is able to detect it. It sends the transaction into a *“cancelling”* state and initiates rollback action. Rollbacks have been designed to be idempotent. On completion of a rollback action, the system changes the transactions state to *“cancelled”*. Also, as shown in [111], the prototype model can also be implemented using other key-value stores.



# CHAPTER 7

## CONCLUSION AND FUTURE WORK

This thesis researched into cloud computing technologies, in general, and NoSQL databases, in particular. NoSQL databases were designed mainly to solve problems of big data such as efficiency, availability and scalability. Though classical relational databases ensure strong consistency of data and support transactions they were found to be inappropriate to meet the requirements of big data. The architectural style and design of NoSQL databases allow them to store large volume of big data and to provide high efficiency, availability and scalability in the processing of big data but at the cost of data consistency and a lack of support for transactions. The research carried out in this thesis has clearly identified the need and importance of transactional support for NoSQL databases. Further, existing research [19] has stressed for the support of transactions in NoSQL databases and the golden standard of ACID properties.

This research addressed the problem of implementing ACID transactions in NoSQL databases. In order to address this problem the research followed appropriate methodological approach.

The challenges with NoSQL databases and their design decisions were thoroughly analysed and understood. The research problems were identified based on an extensive review of existing literature and state-of-the-art systems. With an in-depth understanding and appreciation of the problem area, the research proceeded to propose a solution to implementing ACID transactions in NoSQL databases. The constraints of the proposed systems were clearly defined and specified. Based on these constraints, the prototype system was developed, implemented and tested. The system was then evaluated using standard cloud database benchmarks. The results of evaluation show that the system was able to process transaction efficiently and maintain high level of consistency.

This chapter outlines the contributions made by this research and critically analyses the system and highlights its limitations. Finally, this chapter suggests ways in which the system can be improved. These improvements serve as a basis for future research opportunities.

### 7.1 CONTRIBUTIONS

The following are the contributions made by this thesis.

1. **The definition of a new Multi-Key transaction model for NoSQL systems:** This thesis defined a model for implementing transactions in NoSQL databases (see chapter 4). The approach is different to others in that it does not sacrifice consistency of the system as most NoSQL databases do. The theoretical model for the implementation of this model was defined in [section 4.1](#).
2. **Development of a loosely-coupled architecture that implements transaction logic using a middleware approach:** To be able to achieve the defined multi-key transaction model, this thesis developed and implemented an architecture which contains different components that interact with each other to achieve the defined model. Each of these components has specific duties and functions which were explained in [section 4.3](#).
3. **A new protocol for asynchronous replica management:** Maintaining consistency among replicas of a database is non-trivial. [Section 4.8](#) explains a new protocol for managing replicas designed and implemented in this thesis. The protocol makes use of the Time Stamp Manager (TSM) component of the system to monitor and identify replicas that are stale. The stale replicas are not allowed to be involved in transactions until they are up to date.
4. **The development of new types of operations (read and write) that have stronger consistency guarantee which can be adjusted based on user**

**requirement:** Each of these novel operations allows users/application to determine their level of consistency and latency. This thesis outlined the protocol followed by these operations in [section 5.3.1](#).

5. **The development of a prototype system using real NoSQL system, MongoDB, which is evaluated using the YCSB+T benchmark based on standard Yahoo! Cloud Services Benchmark (YCSB).** The proposed approach (NoSQL-TX) was implemented using cloud technologies and languages which include MongoDB, Python, SQL and JavaScript. The results show enhanced consistency and performance.

The proposed approach is suitable for applications that need transactions and strong consistency. The approach can also work with applications that have interactions between different key items. For instance, an online application that allows multiple users to buy and sell their items. In such applications, there would be interaction among user IDs. Also, the system can be used to manage an online shopping application where users can bid for certain items. In such applications, an item remains available even when a user has put in a bid for it. However, as soon as the user buys the item, it becomes unavailable. The Time Stamp Manager (TSM) can help the system to determine which user has paid for the item first (using the commit timestamp) and then reject all other bids. The TSM can also make bidders aware about the number of bids currently on that system. This is similar to an application that is used to book seats on a flight during checking in. Multiple passengers can see a seat as available until the first passenger books the seat.

## 7.2 CRITICAL ANALYSIS

This section provides a critical analysis of the system being developed in this research. The analysis on one hand provides a critique of the proposed system and on other hand it sets the directions for future work.

First, the proposed system is built around a single NoSQL database, called MongoDB. Though this research provided a justification of the choice for using

## CONCLUSION AND FUTURE WORK

MongoDB, the proposed system could have been validated using some other NoSQL databases. As mentioned earlier, MongoDB is a document class NoSQL database and allows relationships among data entities to be expressed. As a result, the model implemented in this work may not necessarily work with all NoSQL databases. Some NoSQL databases do not allow relationships among data objects to be expressed and this model may not be suitable for such databases. This represents a limitation of the system.

Second, some of the components of the prototype system could have been implemented in a way so that they can cope with increasing number of incoming transactions. For instance, the Time Stamp Manager was not fully optimised to keep up with the speed of incoming requests especially when the number of requests has increased. The reason for this is that before the TSM issues a commit time, it has some processing work to do. It must check for conflicts among ongoing transaction as well as recently completed transaction. Ideally, when two or more transactions approach the TSM at the same time, the TSM tends to queue the transactions. However, when the number of concurrent transactions gets very high, the TSM tends to reject requests. In addition, the TSM could have been replicated in order to survive possible failures. Currently the proposed system uses a single TSM which is susceptible to failures.

To mitigate these limitations, the next section outlines possible solutions that future research can explore.

### 7.3 FUTURE WORK

This section explores the various options that can be implemented to optimise the model of transaction processing proposed by this research. The suggestions are explained below.

**Controller** - Implementing a queuing system that can manage traffic between components of the TSM can improve the transaction throughput (rate of completion) of the system. The requests can be sent through a queue in such a way that as the speed of incoming request increases, the requests are sent

## CONCLUSION AND FUTURE WORK

through queues to prevent loss of transactions. This can have a slight effect on latency but can improve the rate of completion of request. One of such technologies that can be implemented in our system includes rabbitMQ [124]. RabbitMQ is a messaging system that allows applications to connect to each other and scale.

**Optimization of messaging processes** – In implementing transactions, there are message exchanges between components of the system. The process of messaging (and information exchange between the DMS, TPE and TSM) increases latency of operations, particularly write (updates and multi-key) operations. Future research should look into how this information exchange can be optimised in order to reduce the number of network trips.

**Main Memory processing for TSM** – To help the TSM speed up its processing, the TSM can be designed to store and process its data from main memory. This will reduce disk latency and overheads caused by disk I/O. The design of the system included this feature however the prototype system stored its components on hard disk to prevent failures and loss of data. Also, the prototype system was implemented on commodity servers. It is believed that if the time stamp manager (TSM) is implemented on a high performance system, it will improve the throughput and performance of the system.

**Replicating TSM** – As mentioned earlier, the TSM represent a single point of failure for the system. If the TSM should fail, transactions will never be issued with a start-time or commit-time and therefore cannot progress. Future work should consider implementing multiple TSMs to process transactions when the rate of client requests gets very high. The TSM can also be replicated for fault tolerance.

# REFERENCES

- [1] D. W. Cearly and M. J. Walker, "The Top 10 Strategic Technology Trends for 2015," *Gartner Press Release*, 2015. [Online]. Available: <https://www.gartner.com/doc/2966917?srclid=1-3132930191#-1012988714>. [Accessed: 03-May-2016].
- [2] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges," *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7–18, 2010.
- [3] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, p. 50, 2010.
- [4] D. J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, pp. 1–10, 2009.
- [5] P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology," 2011.
- [6] The Open Group, "Service Oriented Architecture." [Online]. Available: <http://www.opengroup.org/subjectareas/soa>. [Accessed: 01-Feb-2016].
- [7] D. Kossmann, T. Kraska, and S. Loesing, "An Evaluation of Alternative Architectures for Transaction Processing in the Cloud," *Proc. 2010 Int. Conf. Manag. data - SIGMOD '10*, p. 579, 2010.
- [8] J. Hamilton, J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a Database System," *J. M. Hellerstein*, vol. 1, no. 2, pp. 141–259, 2007.
- [9] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley

## REFERENCES

- Longman Publishing Co., Inc., 1986.
- [10] R. Smith, R. Harrison, S. Wood, D. Sussman, A. Fedorov, S. Murphy, and Home, *Professional Active Server Pages 2.0*, 2nd ed. Birmingham, UK, UK: Wrox Press Ltd., 1998.
- [11] A. Ogunyadeka, M. Younas, H. Zhu, and A. Aldea, "A Multi-Key Transactions Model for NoSQL Cloud Database Systems," in *IEEE Bigdata2016*, 2016.
- [12] M. Indrawan-Santiago, "Database research: Are we at a crossroad? Reflection on NoSQL," *Proc. 2012 15th Int. Conf. Network-Based Inf. Syst. NBIS 2012*, pp. 45–51, 2012.
- [13] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme, "How to select a Replication Protocol according to Scalability, Availability and Communication overhead," *Proc. 20th IEEE Symp. Reliab. Distrib. Syst.*, 2001.
- [14] L. Lamport, "The Part-Time Parliament," *ACM Trans. Comput. Systems*, vol. 16, no. 2, pp. 133–169, 1998.
- [15] L. Lamport, "Paxos Made Simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 51–58, 2001.
- [16] M. Younas, B. Eaglestone, and K.-M. Chao, "A low-latency resilient protocol for e-business transactions," *Engineering*, vol. 1, no. 3, pp. 278–296, 2004.
- [17] C. Babcock, "Surprise: 44% Of Business IT Pros Never Heard of NoSQL," *Information Week*, 2010. [Online]. Available: <http://www.informationweek.com/database/surprise-44--of-business-it-pros-never-heard-of-nosql/d/d-id/1092523?>
- [18] M. Stonebraker, "Why Enterprises Are Uninterested in NoSQL," *Commun. ACM*, vol. 54, no. 8, p. 10, 2011.
- [19] M. Stonebraker, "SQL databases v. NoSQL databases," *Commun. ACM*, vol. 53, no. 4, p. 10, 2010.
- [20] C. Kothari, *Research methodology: methods and techniques*. 2004.

## REFERENCES

- [21] R. Elio, J. Hoover, I. Nikolaidis, M. Salavatipour, L. Stewart, and K. Wong, "About Computing Science Research Methodology," p. 9, 2011.
- [22] M. Berndtsson, J. Hansson, B. Olsson, and B. Lundell, "Thesis Projects: A Guide for Students in Computer Science and Information Systems," *SpringerVerlag New York, Inc., Secaucus, NJ, USA*, vol. 1, 2007.
- [23] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, vol. 8, no. 4. 2003.
- [24] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*. 2007.
- [25] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 26, no. 6, pp. 64–69, 1983.
- [26] K. P. Eswaran, J. N. Gray, R. a. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976.
- [27] C. Lu, T. Masuzawa, and M. Mosbah, Eds., "Principles of Distributed Systems - 14th International Conference, {OPODIS} 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings," 2010, vol. 6490.
- [28] David M. Kroenke and D. J. Auer, *Database Processing, Fundamentals, Designing, and Implementations*, 12th ed., vol. 1. Pearson, 2012.
- [29] J. Kreps, "The Log : What every software engineer should know about real-time data ' s unifying abstraction," *Linkedin Engineering*, 2013. [Online]. Available: <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>.
- [30] D. O. N. Haderle, "ARIES : A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," vol. 17, no. 1, pp. 94–162, 1992.
- [31] A. Jhingran and P. Khedkar, "Analysis of Recovery in a Database System Using a Write-ahead Log Protocol," *SIGMOD Rec.*, vol. 21, no. 2, pp. 175–184, Jun. 1992.



## REFERENCES

- [32] M. Özsu and P. Valduriez, *Principles of distributed database systems*. 2011.
- [33] J. N. Gray, "Transparency in its Place - The Case Against Transparent Access to Geographically Distributed Data," *Readings Database Syst.*, no. 21667, pp. 592–602, 1994.
- [34] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms, 2/E*. 2007.
- [35] H. Garcia-Molina, J. D. Ullman, and J. Widom, "Database Systems: The Complete Book," *Education*, p. 1248, 2008.
- [36] Gerhard Weikum and G. Vossen, "Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery," *ACM SIGMOD Rec.*, vol. 30, no. 4, p. 853, 2001.
- [37] R. Casado and M. Younas, "Emerging Trends and Technologies in Big Data Processing," *Concurr. Comput. Pr. Exper.*, vol. 27, no. 8, pp. 2078–2091, Jun. 2015.
- [38] P. Zikopoulos, C. Eaton, D. Deroos, T. Deutsch, and G. Lapis, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, vol. 53, no. 9. McGraw-Hill Osborn MEdia, 2012.
- [39] M. Stonebraker and U. Çetintemel, "'One Size Fits All': An Idea Whose Time Has Come and Gone," 2005.
- [40] C. L. Philip Chen and C. Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," *Inf. Sci. (Ny)*, vol. 275, no. January, pp. 314–347, 2014.
- [41] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The End of an Architectural Era ( It ' s Time for a Complete Rewrite )," *Dbms*, vol. 12, pp. 1150–1160, 2007.
- [42] ScaleDB, "Big Data and Transactional Databases Exploding Data Volume is Creating New Stresses on Traditional Transactional Databases." [Online]. Available: <http://docplayer.net/1826199-Big-data-and-transactional-databases-exploding-data-volume-is-creating-new-stresses-on-traditional->

## REFERENCES

- transactional-databases.html. [Accessed: 04-Nov-2015].
- [43] T. Kraska and B. Trushkowsky, "The new database architectures," *IEEE Internet Comput.*, vol. 17, pp. 72–75, 2013.
- [44] R. Hecht and S. Jablonski, "NoSQL evaluation: A use case oriented survey," *Proc. - 2011 Int. Conf. Cloud Serv. Comput. CSC 2011*, pp. 336–341, 2011.
- [45] T. Hoff, "An Unorthodox Approach To Database Design : The Coming Of The Shard," *High Scalability*, 2009. [Online]. Available: <http://highscalability.com/unorthodox-approach-database-design-coming-shard>.
- [46] D. J. Dewitt and J. Gray, "Parallel Database Systems : The Future of Database Processing or a Passing Fad ? conventional shared-nothing hardware base along with a highly parallel dataflow software architecture . Such a design 1 Introduction The 1983 paper titled Database Machines ;," vol. 19, no. 4, pp. 104–112, 1990.
- [47] D. Rubio, "Web Application Performance and scalability techniques," *Web Forefront*, 2013. [Online]. Available: <http://www.webforefront.com/performance/scaling101.html>.
- [48] M. M. Waldrop, "More than Moore," *Nature*, vol. 530, p. 145, 2016.
- [49] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The Dangers of Replication and a Solution," 1996, no. P115, p. 0.
- [50] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo : Amazon ' s Highly Available Key-value Store," pp. 205–220, 2007.
- [51] J. Baker, C. Bond, J. Corbett, and J. Furman, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services.," *Cidr*, pp. 223–234, 2011.
- [52] T. Desai and J. Prajapati, "A Survey of Various Load Balancing Techniques and Challenges in Cloud Computing," *Int. J. Sci. Technol. Res.*, vol. 2, no. 11, pp. 158–161, 2013.

## REFERENCES

- [53] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Proj.* [http://hadoop apache ...](http://hadoop.apache...), pp. 1–13, 2008.
- [54] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, p. 29, 2003.
- [55] E. A. Brewer, "Towards robust distributed systems," *Proc. Annu. ACM Symp. Princ. Distrib. Comput.*, vol. 19, pp. 7–10, 2000.
- [56] A. Fox, S. D. Gribble, and E. a Brewer, "Cluster-Based Scalable Network Services Symposium on Operating Systems Principles," no. October, 1997.
- [57] Laksham Avinash and Prashant Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, pp. 1–6, 2010.
- [58] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, 2002.
- [59] D. Abadi, "Problems with CAP, and Yahoo's little known NoSQL system," 2010. [Online]. Available: <http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>\npapers3://publication/uuid/7223E100-97C1-4BF8-BAF9-EC66435F781F. [Accessed: 20-Apr-2016].
- [60] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.
- [61] D. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *Computer (Long. Beach. Calif.)*, vol. 45, no. 2, pp. 37–42, 2012.
- [62] M. Brooker, "CAP and PACELC: Thinking More Clearly About Consistency." [Online]. Available: <https://brooker.co.za/blog/2014/07/16/pacelc.html>.
- [63] H. Robinson, "CAP Confusion: Problems with 'partition tolerance,'" *Cloudera Engineering Blog*, 2010. [Online]. Available:

## REFERENCES

- <http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/>.
- [64] M. Stonebraker, "Errors in Database Systems, Eventual Consistency, and the CAP Theorem," *Communications of the ACM*, 2010. [Online]. Available: <http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext>.
- [65] E. Brewer, "CAP twelve years later: How the 'rules' have changed," *Computer (Long. Beach. Calif.)*, vol. 45, no. 2, pp. 23–29, 2012.
- [66] P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, 2013.
- [67] W. Vogels, "Eventually Consistent," *Queue*, vol. 6, p. 14, 2008.
- [68] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency Rationing in the Cloud : Pay only when it matters," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 253–264, 2009.
- [69] N. Antonopoulos and L. Gillam, "Cloud Computing: Principles, Systems and Applications," *Media*, vol. 54, p. 379, 2010.
- [70] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," *Proc. - 2011 6th Int. Conf. Pervasive Comput. Appl. ICPCA 2011*, pp. 363–366, 2011.
- [71] D. Agrawal, A. El Abbadi, S. Antony, and S. Das, "Data Management Challenges in Cloud Computing Infrastructures," *Dnls*, pp. 1–10, 2010.
- [72] S. Das, S. Antony, D. Agrawal, and A. El Abbadi, "Clouded Data: Comprehending Scalable Data Management Systems," *Analysis*, no. November, 2008.
- [73] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. a. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *7th Symp. Oper. Syst. Des. Implement. (OSDI '06), Novemb. 6-8, Seattle, WA, USA*, pp. 205–218, 2006.

## REFERENCES

- [74] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," *OSDI '06 Proc. 7th Symp. Oper. Syst. Des. Implement. SE - OSDI '06*, pp. 335–350, 2006.
- [75] J. Dean and S. Ghemawat, "Simplified data processing on large clusters," *Sixth Symp. Oper. Syst. Des. Implement.*, vol. 51, no. 1, pp. 107–113, 2004.
- [76] M. Stonebraker, "The Case for Shared Nothing," *Contract*, vol. 9, pp. 1–5, 1986.
- [77] M. Hogan, "Shared-Disk vs. Shared-Nothing: Comparing Architectures for Clustered Databases by," 2015.
- [78] R. Burtica, E. M. Mocanu, M. I. Andreica, and N. Tapus, "Practical application and evaluation of no-SQL databases in Cloud Computing," *SysCon 2012 - 2012 IEEE Int. Syst. Conf. Proc.*, pp. 79–84, 2012.
- [79] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
- [80] Highly Scalable Blog, "Distributed Algorithms in NoSQL Databases," 2012. [Online]. Available: <https://highlyscalable.wordpress.com/2012/09/18/distributed-algorithms-in-nosql-databases/>.
- [81] R. H. Thomas, "A Majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, 1979.
- [82] C. Plattner and G. Alonso, "Ganymed: scalable replication for transactional web applications," *Proc. ACM/IFIP/USENIX Int. Conf. Middlew.*, pp. 155–174, 2004.
- [83] ORACLE, "Master-slave vs. peer-to-peer architecture: benefits and problems," 2014. [Online]. Available: [https://blogs.oracle.com/NoSQL/entry/master\\_slave\\_vs\\_peer\\_to](https://blogs.oracle.com/NoSQL/entry/master_slave_vs_peer_to).
- [84] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems.," *USENIX Annu. Tech.*, vol. 8, pp.

## REFERENCES

- 11–11, 2010.
- [85] C. Ji, Y. Li, W. Qiu, U. Awada, and K. Li, “Big data processing in cloud computing environments,” *Proc. 2012 Int. Symp. Pervasive Syst. Algorithms, Networks, I-SPAN 2012*, pp. 17–23, 2012.
- [86] MongoDB, “MongoDB Documentation,” 2013. [Online]. Available: <https://docs.mongodb.com/manual/>.
- [87] K. Salem and H. Garcia-Molina, “Main Memory Database Systems: An Overview,” in *IEEE Transactions On Knowledge and Data Engineering*, 1992.
- [88] B. Fitzpatrick, “Distributed Caching with Memcached,” *Linux J.*, no. 124, p. 5–, Aug. 2004.
- [89] D. Obasanjo, “When Databases Lie: Consistency vs. Availability in Distributed Systems,” 2007. [Online]. Available: <http://www.25hoursaday.com/weblog/2007/10/10/WhenDatabasesLieConsistencyVsAvailabilityInDistributedSystems.aspx>.
- [90] P. Sadalage, “NoSQL Databases: An Overview,” *ThoughtWorks*, 2014. [Online]. Available: <https://www.thoughtworks.com/insights/blog/nosql-databases-overview>.
- [91] R. Cattell, “Scalable SQL and NoSQL data stores,” *ACM SIGMOD Rec.*, vol. 39, no. 4, p. 12, 2011.
- [92] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,” in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, 1997, pp. 654–663.
- [93] P. Raman, A. D. George, M. Radlinski, and R. Subramaniyan, “GEMS : Gossip-Enabled Monitoring Service for Heterogeneous Distributed Systems,” *J. Networks, Softw. Tools Appl. Comput.*, vol. 9, pp. 101–120, 2006.

## REFERENCES

- [94] K. K. A and S. Surendran, "BigTable , Dynamo & Cassandra – A Review," *Int. J. Electron. Comput. Sci. Eng.*, vol. 2, pp. 133–141, 2012.
- [95] D. Obasanjo, "Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes," *newtelligence dasBlog*, 2009. [Online]. Available: <http://www.25hoursaday.com/weblog/2009/01/16/BuildingScalableDatabasesProsAndConsOfVariousDatabaseShardingSchemes.aspx>. [Accessed: 12-Feb-2016].
- [96] A. Dey, a Fekete, and U. Röhm, "Scalable transactions across heterogeneous NoSQL key-value data stores," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1434–1439, 2013.
- [97] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner : Google ' s Globally-Distributed Database," *OsdI*, pp. 251–264, 2012.
- [98] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don ' t Settle for Eventual : Scalable Causal Consistency for Wide-Area Storage with COPS Categories and Subject Descriptors," *Lloydia (Cincinnati)*, pp. 1–16.
- [99] D. K. Gifford, "Information Storage in a Decentralized Computer System," Stanford University, Stanford, CA, USA, 1981.
- [100] S. Das and A. El Abbadi, "G-Store : A Scalable Data Store for Transactional Multi key Access in the Cloud," *Group*, pp. 163–174, 2010.
- [101] Z. Wei, G. Pierre, and C. H. Chi, "CloudTPS: Scalable transactions for web applications in the cloud," *IEEE Trans. Serv. Comput.*, vol. 5, pp. 525–539, 2012.
- [102] J. J. Levandoski, "Deuteronomy : Transaction Support for Cloud Data," *Read*, vol. 48, pp. 123–133, 2011.

## REFERENCES

- [103] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications," *Dbms*, vol. 2006, pp. 1–15, 2010.
- [104] F. Junqueira, B. Reed, and M. Yabandeh, "Lock-free transactional support for large-scale storage systems," *Proc. Int. Conf. Dependable Syst. Networks*, pp. 176–181, 2011.
- [105] F. Junqueira and B. Reed, "BookKeeper," *Confluence*, 2016. [Online]. Available:  
<https://cwiki.apache.org/confluence/display/BOOKKEEPER/BookKeeper>.
- [106] P. a. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius, "Adapting microsoft SQL server for cloud computing," *2011 IEEE 27th Int. Conf. Data Eng.*, pp. 1255–1263, 2011.
- [107] S. Harizopoulos and D. Abadi, "OLTP through the looking glass, and what we found there," *Proc. 2008 ...*, vol. pages, p. 981, 2008.
- [108] S. Das, D. Agrawal, and A. El Abbadi, "ElasTraS: An Elastic Transactional Data Store in the Cloud," p. 5, 2010.
- [109] S. Das, D. Agrawal, and A. El Abbadi, "ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud," *ACM Trans. Database Syst.*, vol. 38, no. 1, pp. 5:1–5:45, 2013.
- [110] M. K. Aguilera, M. K. Aguilera, A. Merchant, A. Merchant, M. Shah, M. Shah, A. Veitch, A. Veitch, C. Karamanolis, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," *{SIGOPS} Oper. Syst. Rev.*, vol. 41, no. Figure 1, pp. 159–174, 2007.
- [111] R. Escriva, B. Wong, and E. Sirer, "Warp: Lightweight Multi-Key Transactions for Key-Value Stores," *Rescrv.Net*, 2013.
- [112] A. E. Lotfy, A. I. Saleh, H. A. El-Ghareeb, and H. A. Ali, "A middle layer solution to support ACID properties for NoSQL databases," *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 28, no. 1, pp. 133–145, 2016.
- [113] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling, "Unbundling Transaction



## REFERENCES

- Services in the Cloud,” *arXiv Prepr. arXiv0909.1768*, pp. 1–10, 2009.
- [114] F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey, and S. Thatte, “Web Services Transaction (WS-Transaction),” *IBM Dev.*, p. 25, 2002.
- [115] D. R. K. Ports and K. Grittner, “Serializable Snapshot Isolation in PostgreSQL,” *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1850–1861, 2012.
- [116] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, “Transactional storage for geo-replicated systems,” *Proc. Twenty-Third ACM Symp. Oper. Syst. Princ. - SOSP '11*, p. 385, 2011.
- [117] M. Stonebraker and R. Cattell, “10 Rules for Scalable Performance in ‘Simple Operation’ Datastores,” *Commun. ACM*, vol. 54, no. 6, p. 72, 2011.
- [118] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A Critique of ANSI SQL Isolation Levels,” pp. 1–10, 2007.
- [119] A. S. Dey, “CHERRY GARCIA : TRANSACTIONS ACROSS HETEROGENEOUS DATA STORES,” University of Sydney, 2015.
- [120] Z. Parker, S. Poe, and S. V. Vrbsky, “Comparing NoSQL MongoDB to an SQL DB,” *Proc. 51st ACM Southeast Conf. - ACMSE '13*, p. 1, 2013.
- [121] Y. Lu, “Serializable Snapshot Isolation in Shared-Nothing , Distributed Database Management Systems,” pp. 1–10.
- [122] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” *Proc. 1st ACM Symp. Cloud Comput. - SoCC '10*, pp. 143–154, 2010.
- [123] A. Dey, A. Fekete, R. Nambiar, and U. Rohm, “YCSB+T: Benchmarking web-scale transactional databases,” *Proc. - Int. Conf. Data Eng.*, pp. 223–230, 2014.
- [124] W. Smith, “An Information Architecture Based on Publish/Subscribe Messaging,” in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, 2011, pp. 27:1–27:2.