Data Partitioning and Placement Mechanisms for Elastic Key-Value Stores

Han Li

A dissertation submitted in fulfilment

of the requirements for the degree of

Doctor of Philosophy

The University of New South Wales

School of Computer Science and Engineering

Faculty of Engineering

28 March 2014

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: LI

First name: HAN Other name/s:

Abbreviation for degree as given in the University calendar: Ph.D.

School: School of Computer Science and Engineering Faculty: Faculty of Engineering

Title: Data partitioning and placement mechanisms for elastic key-value stores

Abstract 350 words maximum: (PLEASE TYPE)

Key-Value Stores (KVSs) have become a standard component for many web services and applications due to their inherent scalability, availability, and reliability. Many enterprises are now adopting them for use on servers leased from Infrastructure-as-a-Service (IaaS) providers. The defining characteristic of IaaS is resource elasticity. KVSs benefit from elasticity, when they incorporate new resources on-demand as KVS nodes to deal with increasing workload, and decommission excess resources to save on operational costs.

Elasticity of a KVS poses challenges in allowing efficient, dynamic node arrivals and departures. On one hand, the workload needs to be quickly balanced among the KVS nodes. However, current data partitioning and migration schemes provide low priority to populate new nodes, thereby reducing the effect of adding resources on increasing workload. On the other hand, dynamic node changes downgrade data durability at multiple node failures caused by hardware failure in IaaS Cloud, which is built from commodity components that fail as the norm at large scales; but current replica placement strategies tend to rely on static mapping of data to nodes for high durability.

This thesis proposes a set of data management schemes to address these issues. Firstly, it presents a decentralised automated partitioning algorithm and a lightweight migration strategy, to improve the efficiency of node changes. Secondly, it presents the design of ElasCass, an elastic KVS that incorporates these schemes, implemented atop Apache Cassandra. Finally, it presents a replica placement algorithm with a proof that shows its correctness, to fill the gap of allowing dynamic node changes while maintaining high data durability at multiple node failures. Contributions of this thesis lie in this set of novel schemes for data partitioning, placement, and migration, which provide efficient elasticity for decentralised, shared-nothing KVSs.

The evaluations of ElasCass, conducted on Amazon EC2, revealed that, the proposed schemes reduce node incorporation time and improve load-balancing, thereby increasing scalability and query performance. The other evaluation simulated thousands of KVS nodes and demonstrated that the proposed placement algorithm maintains a close to minimised data loss probability under different failure scenarios, and exhibits better scalability and elasticity than state-of-the-art placement schemes.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

28 March 2014

Signature Witness Date

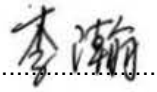
The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

ORIGINALITY STATEMENT

‘I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.’

Signed 

28 March 2014

Date .....

Copyright Statement

Authenticity Statement

Contents

Abstract

Key-Value Stores (KVSs) have become a standard component for many web services and applications due to their inherent scalability, availability, and reliability. Many enterprises are now adopting them for use on servers leased from Infrastructure-as-a-Service (IaaS) providers. The defining characteristic of IaaS is resource elasticity. KVSs benefit from elasticity, when they incorporate new resources on-demand as KVS nodes to deal with increasing workload, and decommission excess resources to save on operational costs.

Elasticity of a KVS poses challenges in allowing efficient, dynamic node arrivals and departures. On one hand, the workload needs to be quickly balanced among the KVS nodes. However, current data partitioning and migration schemes provide low priority to populate new nodes, thereby reducing the effect of adding resources on increasing workload. On the other hand, dynamic node changes downgrade data durability at multiple node failures caused by hardware failure in IaaS Cloud, which is built from commodity components that fail as the norm at large scales; but current replica placement strategies tend to rely on static mapping of data to nodes for high durability.

This thesis proposes a set of data management schemes to address these issues. Firstly, it presents a decentralised automated partitioning algorithm and a lightweight migration strategy, to improve the efficiency of node changes. Secondly, it presents the design of ElasCass, an elastic KVS that incorporates these schemes, implemented atop Apache Cassandra. Finally, it presents a replica placement algorithm with a proof that shows its correctness, to fill the gap of allowing dynamic node changes while maintaining high data durability at multiple node failures. Contributions of this thesis lie in this set of novel schemes for data partitioning, placement, and migration, which provide efficient elasticity for decentralised, shared-nothing KVSs.

The evaluations of ElasCass, conducted on Amazon EC2, revealed that, the proposed schemes reduce node incorporation time and improve load-balancing, thereby increasing

scalability and query performance. The other evaluation simulated thousands of KVS nodes and demonstrated that the proposed placement algorithm maintains a close to minimised data loss probability under different failure scenarios, and exhibits better scalability and elasticity than state-of-the-art placement schemes.

Acknowledgements

This thesis is the culmination of a long journey that would not have been possible without the support offered to me by all the helpful people around me. First and foremost, I would like to thank my principal supervisor, Dr. Srikumar Venugopal, for his advice and ceaseless encouragement throughout the last four years. I am grateful to his devotion to keeping my research career on track. Also, I would like to express my gratitude towards Professor Fethi Rabhi, my co-supervisor, for his support and advice that have helped me move forward through several transitions.

I would like to thank Professor Boualem Benatallah, Associate Professor Wei Wang, and Associate Professor Salil Kanhere, for their serving on my annual progress review committee, and for providing their valuable feedback, which inspired me to carry forward my doctoral study. I would also like to thank Dr. Lawrence Yao and Dr. Nawid Jamali, for their useful advice and encouragement during my doctoral candidature.

I would like to thank the University of New South Wales and the School of Computer Science and Engineering for the research grant, the necessary infrastructure, and the opportunity to pursue my doctoral study. I would like to thank Smart Services CRC Pty Ltd for the grant of Services Aggregation project that made this work possible. In particular, I would like to thank Daniel Austin, Head of Research & Development in Smart Services CRC, for his great effort to provide me a significant amount of data to support my research on distributed data stores. Moreover, I would like to thank Sirca for the help to motivate the application of this research. Also, I would like to Asaf Cidon from Stanford University for his extensive comments on the design of ElasticCopyset that has been incorporated into this thesis.

Last but never the least, I would like to thank my family for their love and support at all times. My mother and father have always been supporting me to climb the mountain of knowledge. They have waited a long time for this. My wife, Yiwen, has made great

sacrifice to leave China and join me. She has been my rock during this journey. I would also like to thank my uncle, Harvey (Weixiong), for his help to my life in Sydney.

List of Publications

Below is a list of relevant publications in which the author appears, sorted in the order of publication date.

1. Venugopal, S., **Li, H**. & Ray, P. 2011, 'Auto-scaling emergency call centres using cloud resources to handle disasters', in *Proceedings of the 19th International Workshop on Quality of Service (IWQoS)*, IEEE Press, San Jose, CA, USA, p. 34.

2. **Li, H**. & Venugopal, S. 2011, 'Using reinforcement learning for controlling an elastic web application hosting platform', in *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)*, ACM, Karlsruhe, Germany, pp. 205–208.

3. **Li, H**. & Venugopal, S. 2013, 'Towards elastic key-value stores on IaaS', in *29th IEEE International Conference on Data Engineering Workshops (ICDEW)*, IEEE, Brisbane, Australia, pp. 302–305.

4. **Li, H**. & Venugopal, S. 2013, 'Efficient node bootstrapping for decentralised shared-nothing key-value stores', in *ACM/IFIP/USENIX International Middleware Conference 2013*, Springer, Beijing, China, pp. 348–367.

List of Figures

List of Tables

List of Acronyms

A list of acronyms used in this chapter is presented as follows in lexical order.

ACID	Atomicity, Consistency, Isolation and Durability
BASE	Basically Available, Soft state, Eventual consistency
CAP	Consistency, Availability, and network Partition tolerance
DFS	Distributed File System
DHT	Distributed Hash Table
IaaS	Infrastructure as a Service
KVS	Key-Value Store
NAS	Network-attached Storage
PaaS	Platform as a Service
P2P	Peer-to-Peer
RDBMS	Relational Database Management System
SaaS	Software as a Service
YCSB	Yahoo! Cloud Serving Benchmark
VM	Virtual Machine

Chapter 1

Introduction

Mighty oaks from little acorns grow.

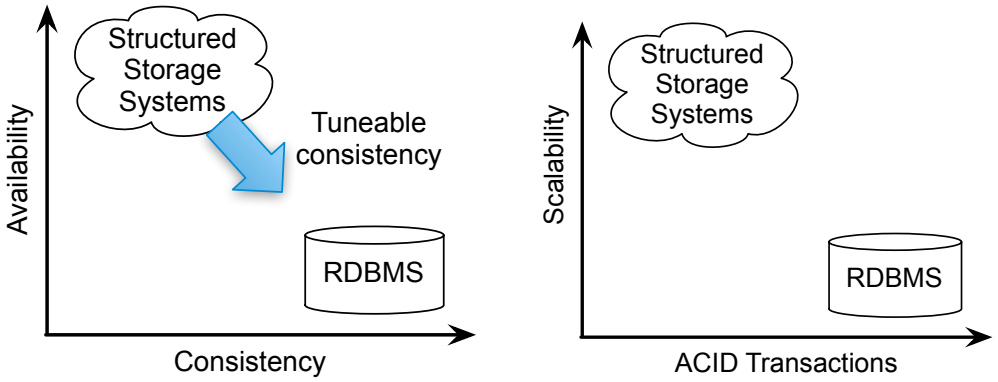
– Alexander Bryan Johnson

This chapter provides an introduction to the research addressed in this thesis. It begins with an overview of distributed data stores, and focuses on the state-of-the-art in Key-Value Stores. Then, it introduces the paradigm of Cloud computing, and discusses the motivation of deploying a KVS on the Cloud. Next, it provides an overview of the research problem, and the objectives of this research. Moreover, it outlines the expected research contribution of this thesis. Finally, this chapter concludes with an outline of the organisation of this thesis.

1.1 Introduction to Distributed Data Stores

Data is central to applications and services. Be it personalised search, sharing information in social networking, or recommending products for online shopping, data plays a key role in improving customer satisfaction. Data drives knowledge, which breeds innovation. Therefore, many modern enterprises are collecting data at the most detailed level possible, resulting in ever-growing data repositories that consist of massive data objects. Hence, software providing the storage and retrieval of data objects forms a critical component of these data-centric stacks. Such software is termed as a **data store** in this thesis.

Thanks to the development of commodity computers, data stores have been designed to run on many low-performance, low-cost machines working in parallel, rather than on individual high-performance, high-cost machines. This has changed the way that a data



(a) The trade-off between availability and consistency, so as to tolerate network partitions

(b) System scalability v.s. transaction support

Figure 1.1: A comparison between RDBMS and structured storage systems

store can grow, in the face of increasing workload demands. For data stores running on multiple commodity computers, the system's capacity is improved by adding more commodity computers. This is termed as **scale-out**. In contrast, for data stores deployed on high-end machines, more resources (such as CPU, memory, and network) are added to a single machine to achieve better performance. This is termed as **scale-up**. Therefore, current-state data stores are distributed systems with the capability of scaling-out. There have been various distributed data stores, classified into two categories: relational database management systems (RDBMSs), and structured storage systems. A description is given as follows.

RDBMSs are classical examples of data stores. They provide rich functionality using the relational model introduced by Codd (1970), along with a declarative query language called SQL. Since the 1980s, these systems have been extremely successful in traditional enterprise settings. One of the key features contributing to the widespread use of RDBMSs is transactional access to data, which is guaranteed by the set of ACID properties, namely *atomicity*, *consistency*, *isolation*, and *durability*. In database transactions, the consistency and integrity of data are treated with the highest priority.

In spite of the success of RDBMSs in classical enterprise infrastructures, their availability and scalability are somewhat limited. On one hand, the issue of availability is codified in Eric Brewer's "CAP Conjecture" (Gilbert & Lynch, 2002), which states that a distributed data store can simultaneously provide only two out of three of the following properties: consistency (C), availability (A), and tolerance of network partitions (P). *Partition tolerance* means that the distributed system continues to operate despite arbitrary

failure of a part of the system. As shown in Figure 1.1a, RDBMSs always choose consistency. Therefore, they sacrifice availability in the face of network partition. On the other hand, since commodity computers are in use, an increase in workload demand requires the system to *scale out* (i.e. adding more commodity computers). However, scaling out RDBMSs has a performance cost, due to the need for distributed lock mechanisms to facilitate transactions. Moreover, during scale-out, it is a major engineering challenge to partition the data following the relational model (Figure 1.1b). Therefore, scalability is more expensive in RDBMSs.

Driven by the demand of real-time web applications and massive data repositories, another class of data stores, called structured storage systems (or *NoSQL*), were designed to provide inherent scalability and finer control over availability. In these systems, data is structured in means other than the relational model as in RDBMSs, so that scalable data partitioning techniques can be employed (Figure 1.1b). In the context of Brewer's CAP Conjecture, structured storage systems do not provide full ACID transaction support, but adopt a weaker consistency model, so as to maintain high availability while withstanding network partitions (Figure 1.1a). Therefore, the varying degrees of weaker consistency engender finer-grained availability.

To meet different demands of applications, structured storage systems have diverse data models, the representatives of which are key-value-based, document-oriented, graph-based, and object-oriented. Amongst the variety of these systems, **Key-Value Stores** (KVSs) have evolved as the most widely used data stores for general-purpose distributed data storage. Examples of KVSs are Google's Bigtable (Chang et al., 2006), Amazon's Dynamo (DeCandia et al., 2007), Yahoo!'s PNUTS (Cooper et al., 2008), and many other open source variants derived from the aforementioned systems. The merits of KVSs include: simple but extendable data model of *key-value* pairs, inherent scalability in data partitioning, and low-latency, highly available data access due to weaker consistency models. The next section will discuss the motivation of designing a KVS for the paradigm of Cloud computing.

1.2 Key-Value Stores on the IaaS Cloud

Cloud computing has emerged as an important paradigm for service oriented computing. It delivers the utilities of hardware and software resources as a service over the Internet.

Analysts project that, worldwide spending on public IT Cloud services is increasing and is expected to exceed over 100 billion dollars in the near future (Gartner, Inc., 2013).

There are three major Cloud abstractions that have gained wide acceptance (Mell & Grance, 2011). First, Infrastructure as a service (**IaaS**) is the lowest abstraction, wherein raw computing resources (such as CPU, memory, storage, and network) are provisioned as a service. Next, Platform as a service (**PaaS**) is a higher service abstraction where an application platform is provisioned as a service. Application developers can develop and deploy customised software solutions on the platform that provides the underlying hardware and software support. Last, Software as a service (**SaaS**) is the highest abstraction layer, which provides the utility of application software and databases as a service.

This thesis focuses on the leverage of IaaS Cloud, which provisions computing resources, usually in the form of virtual machines (VMs), to host distributed data stores (e.g. KVSs). Compared to traditional data centres, the IaaS Cloud brings in new perspectives about utilising the infrastructure. Quoting Armbrust et al. (2010): First, *"the illusion of infinite computing resources available on demand"*; Second, *"the elimination of an up-front commitment by Cloud users"*; Third, *"the ability to pay for use of computing resources on a short-term basis as needed, and release them as needed"*. Moreover, the *transfer of risks* allows small application developers to host systems on servers leased from large infrastructure providers.

Due to these new features, it becomes appealing for KVSs to be deployed on the IaaS Cloud, and more importantly, to add and remove VMs on demand. Note that the workload demands are usually dynamic, caused by orderly human activities or unpredictable events or other factors. Therefore, when the workload demand increases, a KVS is required to *scale out* by adding more VMs as the nodes (i.e. members) of the KVS, so as to maintain a consistent service performance while dealing with extra workloads. In contrast, when the workload demand decreases, a KVS should remove the idle nodes with redundant data off the system (i.e. **scale-in**), so as to reduce the operational cost on the infrastructure, since the IaaS Cloud follows the "pay-per-use" billing model.

Hence, when a KVS is deployed on the IaaS Cloud, it requires that the KVS is capable of matching its capacity to the dynamic workload demands as closely as possible. Such capability is termed as **elasticity**. There are two determinants to the degree of elasticity (Herbst et al., 2013). One is *precision*, which involves an automated controller

that determines when and how many VMs should be added or removed based on the change of workload demands. The other factor is *speed*, which indicates how quickly the KVS can incorporate and utilise the VMs for serving workloads. This thesis targets the latter. Armbrust et al. (2010) had also identified *scaling quickly* as one of the research obstacles/opportunities in Cloud computing. The next section will unveil the research challenges in improving the efficiency of elasticity.

1.3 Research Problem

Resource elasticity is the key feature of the IaaS Cloud, wherein computing resources (e.g. VMs) can be acquired on-demand to deal with increasing workload, and dismissed later to save on operational costs. KVSs benefit from elasticity, when node addition and removal (in other terms, scale-out and scale-in) are as efficient as possible.

However, scaling a distributed data store, such as a KVS, is challenging in that each node is associated with a significant volume of data, which requires delicate design to distribute across a dynamic number of nodes. A good data distribution design can result in well balanced load at each stable scale, leading to optimised resource use, maximised throughput, and minimised response time, without overloading any part of the system.

The strategies of data distribution largely depend on the storage model of a KVS. There have been two models to organise data for a number of nodes: shared-storage and shared-nothing. The **shared-storage** KVSs use a decoupled storage layer, such as networked attached storage (NAS) or distributed file system (DFS), to store a single data copy that is accessed by all the nodes. Thus, adding or removing nodes from a KVS, does not affect the data location on the decoupled storage. Additionally, shared-storage KVSs typically maintain directory services using dedicated components, which are also leveraged to facilitate load balancing and synchronisation in a centralised manner.

In contrast, the **shared-nothing** KVSs are consisting of data nodes, each with their own separate storage that stores one portion of the whole data repository. To avoid data loss due to node failures, each data object has several copies (or replicas) stored by different nodes. Therefore, it is inevitable to reallocate (i.e. move) data replicas across nodes when the nodes are added or removed. However, moving a large volume of data introduces a significant amount of workloads that occupy the resources for serving queries, and thus posing negative impact on query performance. Even worse, shared-nothing KVSs

are typically deployed in a decentralised manner to avoid single-node failure. The lack of dedicated components gives rise to the issue of decentralised coordination.

Hence, this thesis aims to investigate the research problem of how the data can be efficiently distributed for a dynamic number of nodes in shared-nothing KVSs. The goal of efficiency for an elastic KVS is three-fold, and is described as three research problems as follows:

- **Lightweight data movement**. When the nodes are added or removed from a shared-nothing KVS, data movement across nodes is inevitable for the reallocation of data replicas. The design of an elastic KVS must simultaneously address two main, typically opposing, concerns: i) reducing the time required to populate a new empty node with data for serving workloads; and ii) minimising the negative impact of data movement against online query processing. In many KVSs, data movement is usually executed in a low-priority, background thread that can last for many hours (DeCandia et al., 2007). Moreover, due to the dynamic changes in nodes, data movement is more frequent in an elastic KVS. It poses a challenge to ensuring data consistency, while supporting online query processing during data movement.

- **Better load balancing**. The goal of elasticity is to match system capacity against dynamic workloads for better performance and resource utilisation. Such goals can only be achieved when the load that each node undertakes, in terms of both workload demand and data volume, is well balanced at each stable scale after node changes. The challenge of load balancing lies in the design of a more flexible mapping between data and nodes. It also requires a scheme of data storage that quantifies the load shifted in each data reallocation.

- **Higher data durability**. The dynamic nature of elastic KVSs presents a challenge to ensuring high data durability. Maintaining data durability is already difficult in shared-nothing data stores (Borthakur et al., 2011). Note that at a large scale, hardware failure is the norm rather than an exception (Ghemawat et al. 2003, Vishwanath & Nagappan 2010). The failure of hardware components on the IaaS Cloud can cause multiple VMs to fail simultaneously (Guo et al., 2013), which can result in the loss of all the replicas of certain data. Worse still, maintaining data durability is even more difficult in an elastic, shared-nothing KVS, wherein the mapping between

data and nodes is less steady. As a result, high-durability placement strategies relying on static deployment (Cidon et al., 2013) are no longer applicable in elastic KVSs.

In short, these research problems are related to the need for lightweight data movement, better load balancing, and higher data durability in shared-nothing KVSs. Solving these problems will make a shared-nothing KVS *efficiently elastic* to the dynamic workload demands. As can be seen, these problems are all related to the design choices in data management techniques, including partitioning, placement, and movement. However, there is a lack of a set of data management schemes that efficiently deals with node addition and removal in shared-nothing KVSs.

Furthermore, as will be discussed in Chapter 3, most shared-nothing KVSs follow the mantra of "no single source of failure" as in distributed systems. Therefore, the KVSs are typically deployed in a decentralised, symmetric network, where every node plays the same role. This thesis has also adopted this decentralised architecture. That is to say, when it comes to decision-making or distributed synchronisation, there will be no centralised or dedicated components to rely on. Hence, decentralised coordination is another research problem that will be addressed in this thesis.

1.4 Research Objective

The main aim of this thesis is to discover, design, and implement a set of data management schemes that improve the efficiency of elasticity for decentralised, shared-nothing KVSs. The data schemes focus on reducing the time for incorporating new empty VMs as nodes of a KVS, while maintaining online query processing and data consistency during data movement. The data schemes aim to achieve balanced load, in terms of both workload demand and data volume, at each stable scale of the KVS, so as to optimise resource use, maximise throughput, and minimise response time. Moreover, the data schemes address the problem of ensuring high data durability during simultaneous node failures, while allowing dynamic node addition and removal at runtime. Additionally, the data schemes do not assume the existence of any dedicated components; instead, they incorporate the design of decentralised coordination into the execution of distributed tasks.

In order to evaluate the efficiency of the proposed data schemes, it is necessary to im-

plement the proposed schemes into a shared-nothing KVS. The implementation is built on a state-of-the-art open-source KVS project, so as to make use of the advanced techniques that have already been proposed in literature, and to improve the system's performance in general. A widely used benchmark is used to evaluate this implementation on an IaaS Cloud.

1.5 Research Contribution

This thesis makes several contributions towards realising our vision of building elastic KVSs for the IaaS Cloud. The contributions significantly advance the state-of-the-art by supporting efficient and dynamic node additions and removals, while maintaining high query performance and data durability, in decentralised shared-nothing KVSs. These are highlighted as follows.

- This thesis presents a thorough study of the state-of-the-art distributed data stores with the focus on KVSs, and analyse the design choices in building different systems for varying applicability and scope. It then discusses the data management techniques in designing the KVSs that are adaptive on the IaaS Cloud. The objective of this exercise is to carry forward the lessons learned from the rich literature in distributed data stores, and to identify technologies and algorithms developed in related areas that can be applied to the target research area.

- This thesis presents a set of data distribution schemes, to improve the efficiency of adding and removing data nodes in decentralised, shared-nothing KVSs. It reduces the overheads of data movement by consolidating partition replicas into transferable units via automated partitioning, which is accomplished using a decentralised coordination scheme based on election. Moreover, it achieves balanced workload and data distribution using a set of replica placement algorithms that reallocate the partition replicas at node addition and removal. Additionally, the movement of partition replicas is carried out without affecting query performance and data consistency. This set of data distribution schemes fills in the gap of efficient elasticity in decentralised, shared-nothing KVSs.

- This thesis presents the implementation of ElasCass, an elastic, decentralised KVS that handles large volumes of data across many nodes. ElasCass realises the proposed

data distribution schemes to achieve efficient elasticity, and is built atop Apache Cassandra, a widely-used open-source KVS, to provide high performance in terms of scalability and throughput. Compared to current-state KVSs, ElasCass exhibits significantly better resource utilisation in computing and storage due to a well-balanced load, and as a result, delivers a query throughput more than twice that of current-state KVSs. Moreover, ElasCass reduces the time required to incorporate a new node from several hours to within ten minutes, hence fulfilling the goal of efficient node addition.

- This thesis presents ElasticCopyset, a novel data placement scheme that guarantees high data durability while allowing dynamic node changes. Preventing data loss from simultaneous node failures is essential to large-scale systems, wherein hardware failure is the norm rather than an exception (Ghemawat et al. 2003, Vishwanath & Nagappan 2010). In ElasticCopyset, a novel shuffle algorithm is designed, and proved in mathematical terms, to create the minimum sets of nodes for storing the same data, thus achieving a minimised probability of data loss when a certain combination of nodes have failed simultaneously. For example, in a cluster of 5000 storage nodes under a failure event where 1% of nodes are shut down, ElasticCopyset managed to reduce the probability data loss from 99.99% in current-state KVSs to less than 3.8%. Moreover, by supporting a dynamic node number with the use of grouping, ElasticCopyset also exhibits great scalability and elasticity that are not possessed by high-durability, yet static placement strategies (Cidon et al., 2013).

1.6 Thesis Outline

This chapter has provided a general introduction to KVSs and the requirement for elasticity in IaaS Cloud environments. It has also presented an overview of the research question, the research objectives, and the research contributions of this thesis. The remainder of this thesis is organised as follows:

Chapter 2 reveals Cloud computing as an important paradigm for delivering the utilities of computing resources over the Internet, and then discusses the motivation of a key feature of the Cloud, i.e. elasticity. Moreover, it has reviews a variety of distributed data stores in literature, and classifies Key-Value Stores (KVSs) as the state-of-the-art systems

for general-purpose storage and retrieval of data. Finally, it presents the model for hosting a data store (e.g. a KVS) on the Cloud.

Chapter 3 provides a detailed survey of the design space of current-state KVSs under the paradigm of Cloud computing. Next, it examines the schemes of data management in KVSs in terms of node addition and removal for the sake of elasticity. Finally, it identifies the lack of decentralised schemes of data management for the efficient elasticity of KVSs that follow the shared-nothing architecture.

Chapter 4 presents a set of decentralised data distribution schemes to improve the efficiency of elasticity for shared-nothing KVSs. The proposed schemes consist of an automated partitioning algorithm that splits and merges partitions based on the data volume, an election-based coordination to facilitate the decentralised partitioning, a number of replica placement algorithms for node addition and removal, and a data migration policy that ensures online query processing.

Chapter 5 presents the implementation of the set of data distribution schemes proposed in Chapter 4 to build ElasCass. The implementation was developed atop Cassandra (Apache, 2009), and is consisting of three core functionalities, including token management, data storage, and replica reallocation. Chapter 5 also presented the experimental evaluations of ElasCass in several aspects, including data partitioning, node addition, and query performance.

Chapter 6 presents ElasticCopyset, a data placement scheme that maintains the minimum sets of nodes for storing the same data, to minimise the probability of data loss in simultaneous node failures. It also provides a proof for the correctness of the shuffle algorithm in ElasticCopyset. Moreover, Chapter 6 evaluates ElasticCopyset in the scenarios of static deployment, linear scaling, and elastic scaling based on dynamic workloads.

Chapter 7 concludes this thesis with an outline of the thesis contributions, and discusses the limitations and future extensions to the research described herein.

Appendix A contains the mathematical proof for each of the six lemmas of ElasticCopyset defined in Chapter 6.

Furthermore, the core chapters are derived from various articles published during the course of the Ph.D. candidature as detailed below:

Chapter 4 and Chapter 5 are partially derived from:

- **Li, H**. & Venugopal, S. 2013, 'Efficient node bootstrapping for decentralised shared-

nothing key-value stores', in *ACM/IFIP/USENIX International Middleware Conference 2013*, Springer, Beijing, China, pp. 348–367.

Chapter 6 is partially derived from:

- **Li, H**. & Venugopal, S. 2014, 'ElasticCopyset: An elastic replica placement scheme for high durability', Tech. Rep. UNSW-CSE-TR-201402, Computer Science and Engineering, UNSW, Sydney, Australia.

Chapter 2

Background

A rising tide lifts all boats.

– John F. Kennedy

This is the first chapter of the literature review, which is divided into two chapters: Chapter 2 and Chapter 3. This chapter reviews the current literature to examine the key concepts in Cloud computing and distributed data stores, two areas that form the contextual background of this thesis. Next, it discusses the model and requirements for deploying a distributed data store on the Cloud, and then concludes.

2.1 Cloud Computing and Elasticity

2.1.1 Cloud Computing

Cloud computing has emerged as an important paradigm for provisioning the utilities of hardware and software resources over the Internet. There have been various definitions for the phrase "Cloud computing", such as Foster et al. (2008), Buyya et al. (2009), Armbrust et al. (2010). These definitions describe Cloud computing from the perspectives of economic model, system, and datacenter, respectively. In this thesis, the paradigm of Cloud computing is termed as the "Cloud".

This thesis focuses on the leverage of the Cloud to provision computing resources (such as processing, memory, storage, and networking) for hosting data services, and therefore, considers the Cloud as a pool of computing resources. Hence, we follow the definition proposed by Mell & Grance (2011) from NIST (National Institute of Standards and Technology), quoted as:

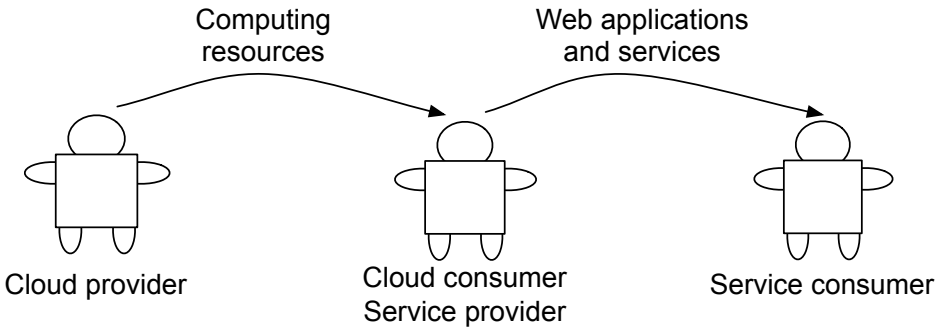


Figure 2.1: Relationships between providers and consumers on the Cloud

- *"Cloud computing is 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."*

Figure 2.1 depicts the relationships between providers and consumers on the Cloud. The Cloud providers offer computing resources, such as servers, networks, and storage, to Cloud consumers. These consumers can be either individual end users who require computing resources, or service providers that leverage these provisioned resources as an infrastructure to deploy web applications and services. Therefore, these service providers are both Cloud consumers and providers, who provide applications and services for the end users, i.e., the service consumers.

Characteristics of Cloud Computing

Buyya et al. (2009) characterised Cloud computing as the model to "deliver computing as the 5th utility" (other than water, electricity, gas, and telephony). Mell & Grance (2011) have summarised five essential characteristics of Cloud computing, but only three of them are closely related to the notion of utility, listed as follows:

- **On-demand provisioning**. A Cloud consumer can acquire and dismiss the resources on a fine-grained, self-service basis near real-time, wherein the resources can be either the utility of hardware such as processor, memory, storage and network, or the services offered by software such as system, platform and application. There are well-established APIs that automatically provision resources as needed, without requiring human interaction with Cloud providers.

- **Resource pooling**. The Cloud provider pools the computing resources via virtualisation of different physical infrastructure, and uses the virtualised resources to serve multiple consumers using multi-tenancy. Hence, each individual consumer can assume that the resource pool is sufficiently large, but should also be aware that the competition between multiple tenants may cause a downgrade in services. For example, the problem of performance interference in the virtualised environments has been well studied (Koh et al. 2007, Nathuji et al. 2010).

- **Metering**. The Cloud provider packages the computing resources as a metered service, wherein resource usage is monitored, controlled, and reported by the provider. The consumer is usually billed with the "charge-per-use" or "pay-as-you-go" model. This model converts capital expenditure to operational expenditure, and thus has the advantage of a low or no initial cost to acquire computing resources.

Cloud computing has many other characteristics, such as rapid elasticity, easy maintenance, low startup cost and broad network access. However, all these characteristics benefit from the notion of delivering computing as a utility, so they are the advance features of Cloud computing, rather than the defining characteristics.

Classifications of Cloud Computing

In Cloud computing, the utility of computing is delivered over the Internet to the consumers as a metered service. To categorise the Cloud, the terminology of "X as a Service (XaaS)" has been widely used (Youseff et al. 2008, Zhang et al. 2010, Mell & Grance 2011), wherein the values of X include Infrastructure, Hardware, Platform, Software, and even Database (Curino et al., 2011). This thesis lists three major Cloud service models that have gained wide acceptance (Mell & Grance, 2011):

- **Infrastructure as a Service (IaaS)** provides the utility of fundamental computing resources, such as processing, storage, and networks, to the Cloud consumers, so that they can run arbitrary software including operating systems and applications. Examples of IaaS providers are Amazon EC2, Rackspace, and GoGrid.

- **Platform as a Service (PaaS)** provides a platform for application developers with the support of programming languages, libraries, services, and tools, so that developers can develop and deploy customised software solutions that can be hosted

by the Cloud platform. Google AppEngine, Microsoft Azure, and Force.com are examples of PaaS providers.

- **Software as a Service (SaaS)** provides the utility of application software and databases that are running on a Cloud infrastructure. The consumers access the services via various client interfaces (e.g. a web browser), and do not manage or control the underlying infrastructure and software. Examples of SaaS providers are Salesforce.com, Oracle's on demand CRM software, and Cumulux.

This thesis has focused on the leverage of the IaaS Cloud, which allows the fully control of the software (e.g. a system that provides data services) on the raw computing resources. Armbrust et al. (2010) also emphasised the utility of Cloud infrastructure, which is further divided into three categories, namely computation, storage, and networking, elaborated as follows:

- **Computation** is typically provisioned in the form of a virtual machine (VM), which is a software-based emulation of a computer that executes programs like a physical machine (Popek & Goldberg, 1974). From a Cloud consumer's view, each VM consists of a static but configurable setup of resources, including processor(s), memory, and "local" storage that is durable only within the VM's lifetime. From a Cloud provider's perspective, each VM is a process (i.e. guest OS) running atop a hypervisor, which uses virtualisation to subdivide and isolate the computing resources from one or several physical machines.

- **Storage** is formulated as a virtualised pool of distributed storage space that is generally hosted in data centres. The storage provider guarantees that the storage is highly fault tolerant and durable, by replicating and distributing the data across many physical storage devices. Similar to computation, Cloud providers virtualise the physical storage and exposes the storage capacity based on the consumers' requirements. The data is stored and retrieved over the network, and can be in various forms depending on the type of storage system running in the backend. For example, Amazon S3 (Amazon, 2006a) stores data objects in the form of binary files, while Amazon EBS (Amazon, 2007) provides raw block devices that can be attached to VMs and accessed via network.

- **Networking** resources include network bandwidth and traffic that are consumed for the communication between computing resources and data transfer between storage. Cloud providers use VLANs and network switches to provide isolated virtual networks. In terms of pricing, Cloud consumers are usually charged by the bytes used, with the rate varying based on the QoS guarantee of the virtual network.

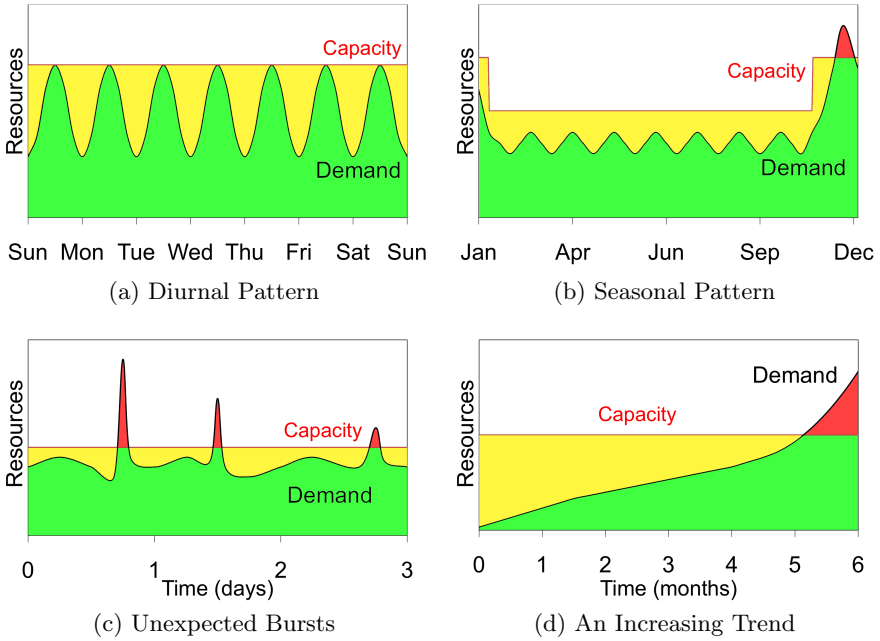
Furthermore, the Cloud can be categorised based on the scope of Cloud consumers (Mell & Grance, 2011). A Cloud is called a *public* Cloud, if it is made available for open use by the general public. Conversely, a *private* Cloud is provisioned for exclusive use by a single organisation, while a *community* Cloud can be shared by a specific community with common concerns. Moreover, there are Cloud consumers who setup a composition of two or more Clouds (private, community or public) to form a *hybrid* Cloud. Note that this categorisation is made regardless of whether the hardware of the Cloud is hosted internally (i.e. on-premise) or externally (i.e., off-premise by a third-party organisation).

2.1.2 Elasticity: A Key Feature of Cloud

Cloud computing introduces the possibility of on-demand provisioning and de-provisioning of computing resources. It brings in new perspectives about utilising the hardware (i.e., computation, storage, and networking). Armbrust et al. (2010) have listed three aspects that are important to the technical and economic changes. First, *"the illusion of infinite computing resources available on demand"*. Second, *"the elimination of an up-front commitment by Cloud users"*. Third, *"the ability to pay for use of computing resources on a short-term basis as needed, and release them as needed"*.

According to these aspects, it becomes possible for systems and applications on the Cloud to add and remove resources commensurately with demand, at a fine grain and in a timely manner. Mell & Grance (2011) defined such feature as *rapid elasticity*, listed as one of the essential characteristics of Cloud computing. Herbst et al. (2013) proposed a definition of elasticity, quoted as:

- *"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible."*



(a) Diurnal Pattern (b) Seasonal Pattern

(c) Unexpected Bursts (d) An Increasing Trend

Figure 2.2: Various patterns of workload demand

The need for elasticity is motivated by variations in workload. There are many factors that cause the demand to change over time. Figure 2.2 depicts a few demand patterns. Firstly, due to orderly human activities, the workloads of web services usually exhibit diurnal and weekly patterns (Atikoglu et al., 2012), shown as in Figure 2.2a. Secondly, annual holidays lead to seasonal or other periodic demand variation in web services. For example, train ticket booking sites peak before holidays, while photo sharing sites peak after holidays; and e-commerce peaks in December (Figure 2.2b). Thirdly, unpredictable events (e.g. news events) cause some unexpected demand bursts (Figure 2.2c). Lastly, the success of a web service results in an increase in its workload and thus the demand (Figure 2.2d). Figure 2.2 also demonstrates that, when the computing capacity is static, the service providers, i.e. computing-resource consumers, have to choose between over-provisioning, which causes waste of resources (marked as yellow), and under-provisioning, which hurts the service performance (marked as red).

Figure 2.3 depicts the ideal scenario where an elastic system adapts its capacity to the changes of workload demand. The benefit of having an elastic system is two-fold. On one hand, when the workload demand increases, the system is able to improve its capacity to deal with the extra demand, by adding more computing resources (i.e. scale out). This ability results in satisfactory service performance. On the other hand, when the demand declines, the system is able to remove the redundant computing resources

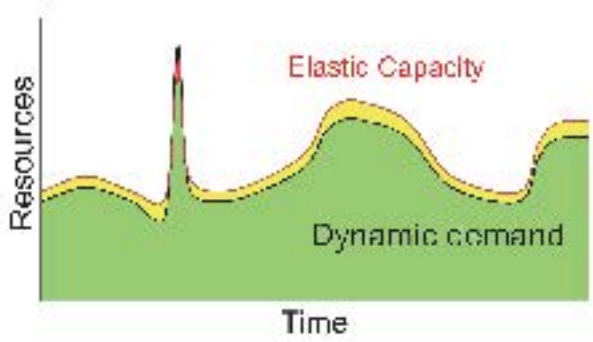


Figure 2.3: Provisioning elastic capacity for dynamic workload demand

that are "charged-per-use" (i.e. scale in). The reward is the reduction in the waste of resources and the economic cost for using Cloud resources.

Up to this point, this section has unveiled the paradigm of Cloud computing, and discussed in general terms the motivation of elasticity, a key feature of the Cloud. The next section will investigate the domain of distributed data stores and their applicability to the Cloud.

2.2 Distributed Data Stores

A distributed data store is a computer network where information is stored, usually in a replicated fashion, on more than one node. This section reviews a variety of distributed data stores in literature.

2.2.1 Distributed Database Systems

Although a *data store* can be referred to any data repository, this thesis reserves the term *databases* for more traditional relational database systems (RDBMS), such as Ceri & Pelagatti (1984), Sheth & Larson (1990). Tamer Özsu & Valduriez (2011) provide thorough surveys of the design space, principles, and properties of databases. These systems are robust in distributing data and query processing over a set of database servers while providing the semantics of centralised systems.

Databases guarantee the reliability of distributed transactions by meeting the ACID semantics (Gray et al., 1981). To elaborate, ACID consists of four properties: **Atomicity** requires that each transaction is either completed or totally aborted (i.e. all or nothing); **Consistency** ensures that any transaction will bring the database from one valid state to another; **Isolation** ensures that concurrent transactions result in a state that would

be achieved if transactions were executed one after another (i.e. in a serialised manner); **Durability** means that a transaction that has been committed will remain so, even in the event of software or hardware failures.

ACID semantics are well suited for commerce transactions in the classical enterprise infrastructure. However, ACID makes no guarantees regarding availability. This issue is codified by Eric Brewer's "CAP Conjecture" (Gilbert & Lynch, 2002), which states that a distributed data store can simultaneously provide only two out of three of the following properties:

- **Consistency (C)** means that all nodes see identical replicas of the data at the same time;
- **Availability (A)** guarantees that every request of an operation receives a response, no matter the execution of the operation is successful or failed.
- **Partition tolerance (P)** means that the distributed system continues to operate despite arbitrary failure of part of the system.

Note that in the face of network partition, a distributed system can retry communication to achieve consistency, using coordination techniques such as Paxos (Lamport, 2001) or two-phase commit protocol (2PC) (Bernstein et al., 1987). These techniques rely heavily on retrying communication and blocking other transactions, which introduce a delay for the data store to make the fundamental decision: either cancel the operation and thus decrease user satisfaction, or proceed with the operation and thus risk data inconsistency. However, availability decreases as the time period of such delay is prolonged, The CAP conjecture points out that, retrying communication indefinitely is in essence choosing consistency over availability. Brewer notes that, to withstand the event of network partition, distributed databases always choose consistency, and sacrifice availability. That is, database systems typically use distributed lock mechanisms to guarantee data consistency, which increases response time of the queries. Therefore, availability is somewhat limited in distributed databases.

Moreover, distributed databases are less flexible in terms of scalability. These systems store tables of data objects that are formally described and organised as the relational model, introduced by Codd (1970). The relational model provides rich functionalities to query data objects that have links (i.e. relationship) between tables, wherein the

four basic relational operators are based on the traditional mathematical set operations, including union, intersection, difference (i.e. except or minus), and cartesian product (i.e. cross join). As can be seen, it is not efficient to execute these relational operations in a distributed manner, because it requires a large number of comparisons and merge of data objects across the nodes. Worse still, when the nodes change (i.e. adding or removing nodes), it is a major engineering challenge to reorganise the data tables (i.e. via database normalisation), while minimising the needs for such set operations. Therefore, in distributed databases, scalability is achieved with considerable administration costs.

Hence, the success of databases is limited in Internet-scale infrastructure, due to their inefficiency in availability and scalability. Particularly, databases are often considered to be less "friendly" to the Cloud environment (Agrawal et al., 2011), because databases are typically intended for an enterprise infrastructure that is statically provisioned, and there is a lack of adequate tools and guidance for databases to scale in and scale out in response to the fluctuation in workloads.

Furthermore, Web applications and services began to serve realtime operations and to consume a large amount of data. The primary value of data stores to the user is not necessarily strong consistency or rich functionalities, but rather high availability of data and high query performance. As a result, there arose research works that have tackled the problem of providing distributed data stores with high availability and scalability over Internet-scale infrastructure. The next subsection describes the evolution of these distributed data stores.

2.2.2 Evolution of Distributed Data Stores

Three representative distributed data stores are reviewed below. These systems are chosen because of their wide acceptance in the Internet-scale distributed environments.

Peer-to-Peer Network

Peer-to-peer (P2P) network (Oram, 2001) is a distributed network architecture, in which interconnected nodes (called *peers*) share resources amongst each other without the use of a centralised administrative component. It builds on the notion that equal peer nodes can function simultaneously as "clients" and "servers" to the other nodes on the network. The benefits of this architecture are: to improve scalability and reliability by removing the

centralised authority, and to ensure redundancy and anonymity so as to tolerate network partitions.

P2P networks have been used for applications in various domains. The most commonly known is content distribution (Ding et al., 2005), covering projects such as Napster (1999), Gnutella (2000), FastTrack (2001), Kazaa (2001), and Freenet (Clarke et al., 2001). Other P2P applications include: instant messaging applications (e.g., ICQ, Yahoo, and MSN), computing resource sharing, such as Seti@Home (Sullivan III et al. 1997, Anderson et al. 2002) and Computer Power Market (Buyya & Vazhkudai, 2001). Recently, there are also P2P-based digital currencies such as Bitcoin (Nakamoto, 2008).

Detailed taxonomies and surveys on P2P networks have been published previously (Milojicic et al. 2002, Androutsellis-Theotokis & Spinellis 2004). These focus mostly on content and file sharing networks that popularised the P2P technology. The focus of such networks is on designing efficient strategies to locate particular content amongst the peers, and to transfer content reliably even in the face of high volatility. The early-stage P2P systems are mostly unstructured, in which the placement of content (e.g. files) is completely unrelated to the overlay topology, that is, the network topology of how the nodes are overlaid on the Internet. As a result, these systems suffer severe scalability and performance problems, because there is a lack of mechanism to balance the workload and data distribution as the nodes join or leave. This issue motivated the development of structured P2P network, described in the following subsection.

Distributed Hash Tables

Distributed Hash Tables (DHT) are structured P2P networks (Lua et al., 2005), wherein the overlay topology of nodes is tightly controlled and data objects (or pointers to them) are placed at precisely specified locations, which makes query routing more efficient.

In DHT-based systems, data objects (i.e., *values*) are assigned unique identifiers called *keys*, which are mapped by the overlay network protocol to a unique peer in the network. Each peer has a unique NodeID, and maintains a small routing table consisting of its neighbouring peers' NodeIDs and IP addresses. In this way, the responsibility for maintaining the mapping from keys to data objects is distributed among the nodes, which ensures minimal amount of disruption when node changes occur. Hence, DHT-based systems are capable of scaling to extremely large numbers of nodes and dealing with arbitrary node

arrivals and departures.

There are various DHTs that use different schemes for data objects, key space, and routing strategies. The four best-known DHT systems are: Content Addressable Network (CAN) (Ratnasamy et al., 2001) that is architected as a virtual multi-dimentional Cartesian coordinate space on a multi-torus; Chord (Stoica et al., 2001) that leverages consistent hashing (Karger et al., 1997) to form uni-directional and circular key space; Tapestry (Zhao et al., 2001, 2004) and Pastry (Rowstron & Druschel, 2001) that both use variants of Plaxton mesh network (Plaxton et al., 1999) for load distribution and routing locality. Other popular DHTs include Kademlia (Maymounkov & Mazieres, 2002), Viceroy (Malkhi et al., 2002), Koorde (Kaashoek & Karger, 2003) and BitTorrent (Cohen, 2003).

In theory, given N peers in the system, all of these DHTs guarantee that any data object can be located in $O(\log N)$ overlay hops on average. Moreover, these systems minimise the disruption caused by node joining or leaving, by confining each node to coordinate with only $O(\log N)$ other nodes.

One major limitation of DHTs is that they only support exact-match lookups: one needs to know the exact key of a data object to locate the nodes that store it. This has led to research on building a distributed query engine on top of a DHT. SkipNet (Harvey et al., 2003) supports logarithmic time range-based lookups and guarantees path locality, by organising peers and data objects according to their lexicographic addresses in the form of a probabilistic skip list. PIER (Huebsch et al., 2003) runs relational queries across thousands of machines, with a relaxed consistency called "dilated-reachable snapshot". Gupta et al. (2003a) used locality sensitive hashing to find data ranges for approximate answers to the complex queries. Mercury (Bharambe et al., 2004) is a scalable protocol that supports multi-attribute queries and explicit load balancing.

Nonetheless, the focus of P2P systems (including DHTs) is on providing a lookup service that efficiently retrieves the value associated with a given key. They do not explicitly deal with data consistency, and therefore, do not settle down the trade-offs between consistency, availability, and network partition tolerance discussed in Brewer's CAP Conjecture (Gilbert & Lynch, 2002). This leads to the introduction of structured storage.

Structured Storage

With the need for scaling real-time web applications and the increase in the amount of available data, there arose the demand for data stores that are intended primarily for simple retrieval and appending operations, while possessing the properties of high horizontal scaling, high availability, and high performance in serving queries. Systems with such characteristics are termed as structured storage, which provide various mechanisms for storage and retrieval of data that is structured in means other than the relational model (Codd, 1970) in databases.

Structured storage typically process queries with the requirement of BASE (Fox et al., 1997), a data semantic that is weaker than ACID (Gray et al., 1981) as in databases. In the parlance of Eric Brewer's CAP Conjecture (Gilbert & Lynch, 2002), systems following ACID semantics always choose consistency over availability. In contrast, the BASE semantics trade consistency for availability and rely on soft state for robustness in failure management. The three properties of BASE are described as follows:

- **Basically Available (BA)** means that the service as a whole must be available during runtime, despite transient partial hardware or software failures;
- **Soft state (S)** means that, stale data can be temporarily tolerated, and can be regenerated at the expense of additional computation or disk I/O;
- **Eventual consistency (E)** guarantees that if no new updates are made, all copies of the data will eventually reach consistency after a short time.

The essence of BASE is to allow soft state, i.e., the temporary existence of stale data. Compared to ACID, there are two advantages of BASE. First, it improves performance from avoiding transaction commits that require blocking, which affects system availability. Second, it increases robustness in the event of network partitions, because it allows to postpone communication and disk I/O until a more convenient time, rather than requiring durable and consistent state across partial failures as in ACID semantics.

Moreover, eventual consistency "guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value" (Vogels, 2009). This weak consistency model makes distributed data stores tolerable to network partition, in the Internet-scale environment where "component failures are the norm rather than the exception" (Ghemawat et al., 2003). Moreover, structured storage allows SQL-like query

languages to be used, and therefore, is also referred to as "NoSQL", which is currently read as "Not only SQL".

Structured storage systems are diverse in data model and system architecture, and are designed to meet demands of different applications. There have been various approaches to categorising such data stores. Here presents a classification of structured storage based on the modelling of data. The most popular data models (DB-Engines, 2013) are key-value based, document-oriented, and graph-like, as described as follows:

- **Key-Value Store (KVS)** enables applications to store schema-less data in the form of a *key-value* pair, usually consisting of a string representing the *key*, and the actual data considered to be the *value*. This key-value mapping can be recursive, such that the value of a data object can be a collection of key-value pairs. As this data model closely resembles DHTs' key-value mapping, KVSs can inherit the network architectures developed in DHT systems. Hence, KVSs also possess high performance and scalability in storing and retrieving data indexed by unique identifiers. Examples of KVSs include Memcached (2004), Google's Bigtable (2006), Amazon's Dynamo (2007), Apache Cassandra (2009), and Redis (2009).

- **Document Store** is also known as document-oriented database. Document stores are designed for storing, retrieving, and managing semi-structured data (Abiteboul et al., 2000) in the form of a *Document*, which encapsulates and encodes data in standard formats such as XML, YAML, JSON, and BSON. Documents are addressed via a unique key that is often a simple string, a URI, or a path. The advantage of document stores is that they offer API or query language that facilitates the retrieval and integration of documents based on the content. Current-state document stores are Apache CouchDB (2008a), MongoDB (2010), and Couchbase (2012).

- **Graph Database** builds on graph theory (Bondy & Murty, 1976) to store and provide efficient queries on data sets in the associative model (Williams, 2000), in which everything is modelled as an *entity* and *associations* between the entities. Graph databases employ graph structures with nodes, properties, and edges to represent data. Entities are represented as nodes, while the pertinent information related to nodes are properties. The associations from nodes to nodes or from nodes to properties are represented as edges. Hence, graph databases are a powerful tool

for graph-like queries. Popular graph databases are Neo4j (Neo Technology, 2010) and InfiniteGraph (Objectivity Inc., 2010).

There are other structured storage systems. Triple stores (a.k.a. RDF stores) (Wilkinson et al., 2003) manage triples in the domain of semantic web (Berners-Lee et al., 2001), wherein a triple is a composition of subject-predicate-object, and is usually expressed using Resource Description Framework (RDF) (Klyne et al., 2004). Object databases, such as Db4o (Versant corporation, 2008) and ObjectStore (Lamb et al., 1991), represent information in the form of objects as used in object-oriented programming.

Among the variety of structured storage, Key-Value Stores (KVSs) have evolved as the most influential data store for general-purpose distributed storage, because of the simplicity in terms of data model and operation, optimised performance in query processing, and high availability and scalability. Current-state KVSs will be revealed in the next chapter.

Comparisons of Various Data Stores

This section has introduced four categories of distributed data stores. RDBMS have been a predominant choice for the storage of relational data and the support of transactional queries. P2P networks were developed to allow individual Internet end users to share files amongst each other. Distributed Hash Tables (DHTs), as structured P2P networks, support more complex services with better routing performance than traditional P2P networks. Structured storage systems have become a standard reference for serving data for web applications that require high performance data retrieval.

Table 2.1 summarises these data stores. Aside from the purpose and examples, there are five properties chosen to differentiate these data stores. From the perspective of infrastructure, P2P and DHT systems focus on key lookups based on the overlay of nodes in a network, while RDBMS and structured storage are typically deployed in a cluster of interconnected commodity (or virtual) machines to serve more sophisticated queries. In terms of architecture, DHTs follow a structured overlay of nodes to provide key-based routing that is more efficient than that of P2P networks, which use unstructured overlay.

Moreover, RDBMS and structured storage differ in their data model and consistency model. That is, RDBMS store data in the relational model (Codd, 1970), and support transactional queries by meeting the ACID semantics (Gray et al., 1981). In contrast, structured storage avoid the use of the relational model to simplify the partitioning of data,

Property	**RDBMS**	**Peer-to-Peer Network**	**Distributed Hash Tables**	**Structured Storage**
Purpose	Store relational data, support transactional queries	Share files across Internet	Support more complex services with structured networks	Serve big data and real-time web services and apps
Infrastructure	Interconnected machines on traditional enterprise infrastructures	Enormous end devices across Internet	Interconnected end devices with identifiers	Interconnected machines across wide area network
Architecture	Multiple masters or master-slave	Unstructured P2P network	Structured P2P on overlay network	Either centralised or decentralised
Data Model	Relational model	Arbitrary files	Key-value pairs	Key-value, document, graph, object, etc.
Consistency Model	ACID semantics	Weak	Weak	BASE semantics
Data Discovery	B tree indexing	Flooded requests or document routing	Key lookup within O(log n) hops	Key mapping in 0-hop
Examples	Oracle DB, MySQL, IBM DB2, and MS SQL Server	Napster, Gnutella, Kazaa, and Freenet	Chord, CAN, Tapestry, and Pastry	Bigtable, Dynamo, MongoDB, and Neo4j

Table 2.1: Comparison between various distributed data stores

and serve simple retrieval and appending operations by following the BASE semantics (Fox et al., 1997).

In addition, these four types of data stores are also varied in the way of data discovery. RDBMS use many variants of Binary tree to build indexes for the keys of the data, while DHTs and structured storage rely on hash functions to maintain the mapping between data and nodes. By comparison, unstructured P2P networks adopt a flooding query model, which is the least efficient in these four data stores.

2.2.3 Distributed Data Stores on the Cloud

Subsection 2.1.1 presents a number of classifications of the Cloud (page 15). Based on these classifications, Figure 2.4 presents a typical model for the leverage of IaaS Cloud to

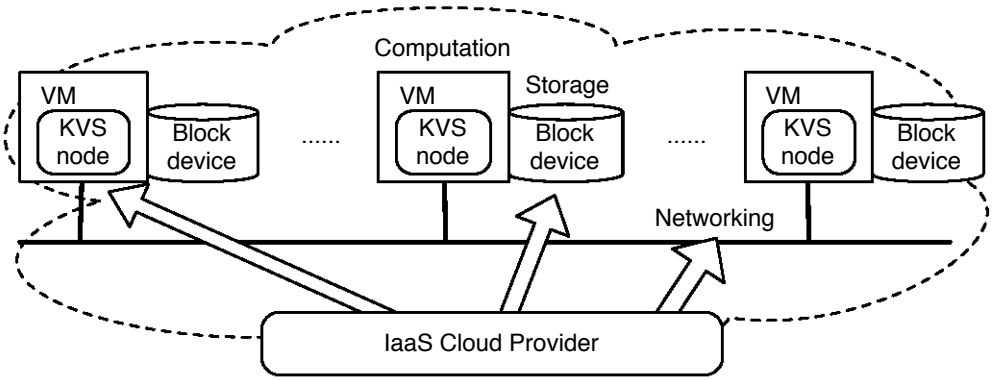


Figure 2.4: A typical model to deploy a distributed data store on the IaaS Cloud

host a distributed data store (e.g. a KVS).

As can be seen, the provider of IaaS Cloud, either public or private, provisions the computational capacity in the form of multiple VMs, each hosting one node (or one instance) of the distributed data store. Since VMs normally do not provide durable disks, the persistence of data is guaranteed by the storage provisioned from the Cloud. Although the storage space can be of various forms, the widely accepted form of storage is raw block device (e.g. Amazon EBS), because it closely resembles the hard disk of a commodity machine, and can be attached to the VMs at runtime, serving like a local disk. In addition, networking is usually provisioned along with the VMs, and serves as a means to data delivery across the VMs.

This model provides data services using a a stateless computation tier, along with a stateful storage tier. It is widely accepted in the industry (Cockroft, 2011), and is also adopted in this thesis due to two considerations. First, each VM, when attached with a storage device, closely resembles a commodity machine, so that systems running in traditional data centres can be seamlessly migrated to the Cloud. Second, decoupling the resources of computation, storage, and networking, allows us to identify new challenges that are not presented in computing environments composed of commodity machines.

Requirements for Elastic Data Stores

In the early stage of Cloud computing, the elasticity feature has been exploited by stateless applications and services, and is termed as "auto-scaling". For example, AppEngine (Google, 2008) provides automatic scaling feature for web applications, by enforcing an application structure of clean separation between a stateless computation tier and a stateful storage tier. Stateless applications and services are the ideal model for scaling, be-

cause adding and removing computation nodes (which are decoupled from storage space) is sufficient to track workload variation.

In contrast, it is more challenging to achieve elasticity in a data store. Even though computing resources can be provisioned on demand from the Cloud, one should be aware that, adding or removing certain amount of computing resources, does not necessarily increase or decease the capacity of a data store. The reason is that, each storage node has to acquire control of its own data before it can serve queries, but handing over a large amount of data to a new, empty node requires significant amount of workloads, and more importantly, the delicate design of schemes to redistribute the data across the nodes.

To enable elasticity, there are two requirements for data stores (and many other systems) to convert the computing resources into system capacity (Herbst et al., 2013):

- **Precision** is the deviation between the system capacity and the current workload demand. Since computing resources can be provisioned (and charged) at a fine-grained level, the challenge of precision lies in resource management strategies that control the provisioning of resources based on the change of workloads.

- **Speed** is the time period from the state where resource provisioning is in need, to the state where the system's capacity has matched up to the demand. Cloud providers can allocate new VMs quickly within several minutes (Li et al., 2010), so the challenge of speed for a data store is how quickly it can incorporate and utilise the VMs for serving workload demand. The speed of scaling a data store is determined by the data management schemes that partition, replicate (or place), and migrate the data over the cluster of storage nodes.

Accordingly, the requirements of precision and speed in terms of scaling a data store, lead to two research problems: resource management, and data management.

The focus of resource management for elastic data stores is on designing an automated controller, which continuously monitors the system and automatically determines *when* and *how much* to provision (or de-provision) resources. There have been research on resource management. Lim et al. (2010) identified the constraint of actuator delays in the controller, which is caused by data rebalancing (i.e. the redistribution of data across nodes), and proposed a cost-optimisation-based approach to determine the amount of bandwidth to use for rebalancing data, and to trade the impact on the guest service against

the time to complete data rebalancing. The SCADS Director (Trushkowsky et al., 2011) is a control framework for Key-Value Stores (KVSs). It uses a machine learning model (i.e. Model-Predictive Control) to predict resource requirements based on workload statistics, and to move and replicate data as needed. SCADS also organises data as small units for a finer granularity of load-balancing and workload statistics. EStoreSim (Moulavi et al., 2012) is a simulation framework that allows developers to simulate an elastic KVS on the Cloud, and be able to experiment with different controllers and workloads.

These publications investigated meeting service level objectives (SLOs) and reducing costs by adding or removing computing resources. However, none of them provide discussions on how the data can be distributed when the number of nodes (that store the data) changes, because the schemes of data management vary drastically in different categories of data stores. The next chapter will focus on KVSs, the most influential data stores, and discusses the approaches towards elasticity in current-state KVSs.

2.3 Chapter Summary

This chapter has revealed Cloud computing as an important paradigm for delivering the utilities of computing resources over the Internet. It has reviewed the three Cloud service models that gained wide acceptance (Mell & Grance, 2011), namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), and identified IaaS as the primary model for hosting data stores. Next, it presented the definition and motivation of a key feature of the Cloud, i.e. elasticity, which requires a system to adapt its capacity to workload changes for better performance and resource utilisation.

Moreover, this chapter has reviewed four categories of distributed data stores. It started by discussing traditional databases (i.e. RDBMS) and their limitations in scalability and availability due to the constraint of ACID semantics (Gray et al., 1981) and the relational model (Codd, 1970). In light of Eric Brewer's "CAP Conjecture" (Gilbert & Lynch, 2002), and with the need for scaling real-time web applications and serving an increasing amount of data, there arose many distributed data stores that loosen the ACID constraint and focus on scaling over the Internet. Three categories of distributed data stores have evolved since then, and structured storage is the current-state data store that focuses on high horizontal scaling, high availability, and high performance in serving quer-

ies. Amongst the variety of structured storage, Key-Value Stores (KVSs) have risen to be the state-of-the-art systems for storage and retrieval of data for general purposes.

Given the background of Cloud computing, elasticity, and distributed data stores, this chapter presented the model for hosting a data store (e.g. a KVS) on the IaaS Cloud. It also discussed the requirements for achieving elasticity in a data store, which leads to the challenges in data management for dynamic node addition and removal. The literature review continues in the next chapter, which will examine this question in the context of KVSs.

Chapter 3

State of the Art

If I have seen further it is by standing on the shoulders of giants.

– Isaac Newton

This chapter is the second and final chapter of the literature review. The previous chapter has concluded that Key-Value Stores (KVSs) are the state-of-the-art data stores for general purposes, and has also discussed the model and motivation for designing an elastic data store using the IaaS Cloud. The purpose of this chapter is to review existing approaches to building an elastic KVS that is able to adapt its capacity to the workload changes by efficiently adding or removing nodes. This chapter begins with a thorough survey of the design choices in KVSs. Then, it discusses the existing schemes of data management for the sake of building an elastic KVS. Finally, this chapter identifies the research gaps and then concludes.

3.1 Key-Value Stores (KVSs)

KVSs have become "a high-performance alternative to relational database systems with respect to storing and accessing data" (Seeger, 2009). Ever since the advent of the first generation KVSs, such as Memcached (Fitzpatrick, 2004), Bigtable (Chang et al., 2006), Dynamo (DeCandia et al., 2007), and PNUTS (Cooper et al., 2008), KVSs have been evolving in both quality and quantity.

As categorised in Figure 3.1, these KVSs differ in several fundamental aspects in their designs, including system architecture, data consistency model, system quality attributes, and query performance. This section presents a thorough survey of current-state KVSs,

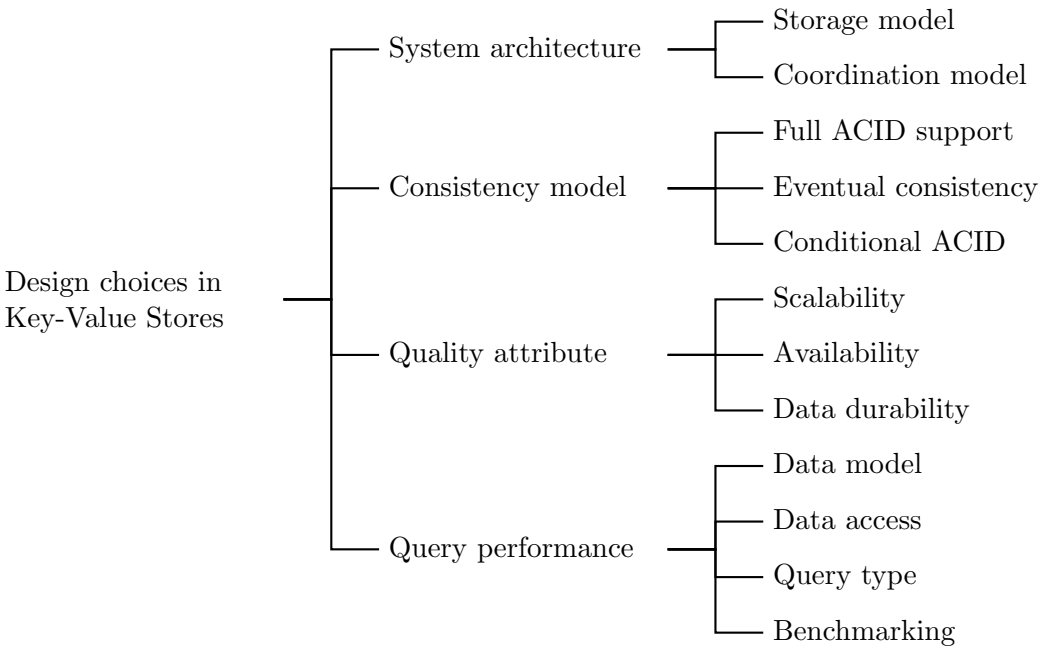


Figure 3.1: Categorise the design choices in Key-Value Stores

by discussing the choices made in the design of KVSs. It also introduces a number of state-of-the-art KVSs that have significant influence to the development of KVSs.

3.1.1 System Architecture

The design choices in system architecture separates a KVS from one category to another. The storage model determines the performance and approaches to data management strategies, while the coordination model establishes the way how the nodes of a KVS collaborate with each other. KVSs exhibit various characteristics due to the choice of different storage and coordination models.

Storage Model

Figure 3.2 provides a classification of storage models based on the storage device that the data is mainly hosted. Storage device is used as the classification criteria, because the performance of data access differs drastically using different devices. The most traditional device is hard disk, on which there are two storage models:

- **Shared-storage** treats the whole data set as one single large data image, which stored in separate location, such as networked attached storage (NAS) (Gibson & Van Meter, 2000) or distributed file system (DFS) (Ghemawat et al. 2003, Shvachko

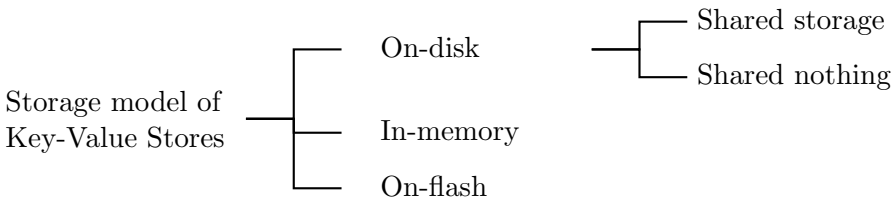


Figure 3.2: A classification of Key-Value Stores based on storage model

et al. 2010) that can be accessed by all the nodes of the KVS. Hence, when a node is added (or removed) to the KVS, only the metadata (e.g. indexes and file pointers) is required to be migrated across nodes, while the actual data remains unchanged in the common storage space. Examples of shared-storage KVSs are Bigtable (Chang et al., 2006), HBase (Apache, 2010), and Hypertable (Hypertable, 2009).

- **Shared-nothing** requires the partitioning of data into independent sets that are physically located on different nodes. Each node is self-sufficient in maintaining a portion of the key space and the related data on their own storage, without sharing memory or disk storage with any other peer nodes. Hence, data migration is inevitable when a node is added or removed. Dynamo (DeCandia et al., 2007), PNUTS (Cooper et al., 2008), Cassandra (Apache, 2009), and Project Voldemort (2009) are examples of shared-nothing KVSs.

Aside from the use of hard disks, there are many KVSs using RAM or flash memory as the storage media, termed as in-memory and on-flash KVSs, respectively.

In-memory KVSs keep the entire data sets exclusively in the RAM devices in a distributed fashion. The in-memory nature allows these systems to support extremely high performance beyond the limitation of slow I/O operations using hard disk. Since RAM devices are volatile, there are two classes of in-memory KVSs designed to meet different functionality requirements. One is used as a memory caching system in front of a database, to improve performance and offload the backend system. An example is Memcached (Danga Interactive, 2004). The other class of in-memory KVSs is a stand-alone KVS that that requires external storage to provide data durability, like Redis (2009). However, due to the high cost per bit of RAM, in-memory cache and storage are still an expensive approach to serving billions of key-value objects. In-memory KVSs are either constrained by the infrastructure to store limited amount of data, or require a costly expenditure on RAM devices.

With the increasing demand of high-performance data-intensive applications, recent trends have been to design KVSs using a combination of RAM and flash memory, a persistent storage medium in the form of solid state disks (SSDs). Flash memory stands in the middle between RAM and hard disk in terms of cost and performance. It is 10x cheaper than RAM, while 20x more expensive than disk. According to Leventhal (2008), flash provides access times that are 100-1000 times lower than disk while 100 times higher than DRAM. There are two types of flash devices, namely NOR and NAND flash. The latter allows a denser layout and greater storage capacity per chip, and therefore, is widely used for general storage.

On-flash KVSs store key-value pairs fully on flash memory, while maintaining a small size of metadata per key in RAM to support faster data lookups. These KVSs focus on reducing the size of metadata per key stored in memory, as well as the number of accesses to flash memory required per key-value pair. For example, SkimpyStash (Debnath et al., 2011) reduces the memory footprint to about 1 byte per key, compared to 6 bytes per key in FAWN (Andersen et al., 2009) and ChunkStash (Debnath et al., 2010a). Moreover, SILT (Lim et al., 2011) achieves 1.01 flash reads for key-value pair with even less memory footprint, in contrast to SkimpyStash that requires five flash reads per lookup. Despite that there are KVSs using flash memory as the secondary *cache* in the middle of RAM and disk, e.g. FlashStore (Debnath et al., 2010b), the overall trend is to replace hard disk with flash memory as a general-purpose storage media in the future. In this sense, the storage models for hard disk (i.e. shared-storage and shared-nothing) can be inherited by on-flash KVSs.

Table 3.1 summarises the storage models described. On-disk KVSs are the most widely used because of low cost of hard disks, but they also exhibit the slowest data access out of the three classes. In-memory KVSs go to extremes by hosting the entire data set in memory for outstanding performance, but pay the price of high cost in storage and extra support for data durability. By comparison, on-flash KVSs stand between on-disk and in-memory. The decreasing price of flash memory makes it a good replacement for the hard disk drive in the near future.

Property	On-disk	In-memory	On-flash
Storage Device	Hard disk drive	DRAM / RAM	SSD, typically NAND-based flash memory
Cost	Lowest cost	Highest cost	Medium cost
Performance	Slow in both read and write	Best in both read and write	Better in read than write, bad at small writes
Strength	Inexpensive, vast storage space	Extremely high query performance	Good balance between cost and performance
Weakness	Disk I/O is expensive	High cost per bit, extra concern on durability	Cost is higher than disk, but decreasing
Examples	Bigtable, Dynamo, Cassandra	Memcached, Redis	SkimpyStash, SILT

Table 3.1: Summary of various storage models of Key-Value Stores

Coordination Model

KVSs are distributed systems, consisting of a large number of interconnected nodes. The relationships between nodes and the roles each node plays are diverse in different models. There are two common coordination models: master-slave (i.e. centralised), and peer-to-peer (i.e. decentralised).

In the master-slave model, a master is elected from a group of eligible nodes, to facilitate metadata management and decision making, while the other nodes are acting in the role of slaves. For example, Bigtable (Chang et al., 2006) and PNUTS (Cooper et al., 2008) use one master (or controller) per region to support geographically distributed data stores. The use of a centralised master in each region makes it easier to implement sophisticated data placement and replication policies, as the master possesses and controls most of the relevant information. The key design related to this model is the election of a master, which is usually based on distributed consensus protocols (Lamport, 1998). For example, Bigtable relies on distributed lock service called Chubby (Burrows, 2006), in which the master must obtain votes from a majority of the replicas (i.e. slaves), each promising that they will not elect a different master for a time interval. This promise is called the master lease, which is periodically renewed by the replicas.

The peer-to-peer model is typically based on decentralised DHT networks. Examples are Dynamo (DeCandia et al., 2007), Cassandra (Apache, 2009) and Project Voldemort

(2009). These systems are able to scale to extremely large numbers of nodes, and more importantly, are capable of handling continual node arrivals, departures, and failures without any dedicated component. The key design related to decentralised KVSs is message exchanging for routing information and membership. The typical choice is to use the gossip-based protocols (Lin & Marzullo, 1999), where each node, on receiving a message, forwards the message to a small subset of nodes that are randomly selected. This gossip-based messaging builds on probabilistic broadcast algorithms that trade reliability guarantees against scalability.

A Comparison of System Architectures

This thesis follows the use of the most traditional storage devices, i.e., hard disks, to store data of a KVS, because our aim is to provide data services for general purposes. Hence, in the remaining of this chapter, the focus is on the on-disk KVSs. As discussed, the on-disk model is classified into shared-storage and shared-nothing. Table 3.2 presents a detailed comparison between these two storage models, with extra considerations on the coordination model.

Shared-storage KVSs typically use dedicated directory services to improve flexibility in the mapping between data objects and nodes. Hence, they commonly leverage the same dedicated component to act as a master node. The advantage of this **centralised, shared-storage** architecture is three-fold. First, the stateless computation is separated from the stateful storage. That is, the reliability and scalability of data are maintained by the underlying NAS or DFS, so that KVS nodes, which are responsible for query processing, can be added or removed without moving the actual data across nodes. Second, due to the existence of dedicated directories, it allows dynamic reallocation of data to achieve better load balancing. Third, the master node can simplify the coordination between nodes.

However, the dependence on dedicated components also gives rise to several disadvantages: i) it makes the system more vulnerable to single-node failure; ii) the availability of the system depends on the functioning status of the dedicated nodes; and iii) the total number of nodes in the system is constrained by the capability of the dedicated nodes, hence the scalability is somewhat limited.

In contrast, shared-nothing KVSs typically partition the key space into a structured P2P overlay topology to essentially form a DHT. Hence, this **decentralised, shared-**

Property	**Centralised, shared-storage**	**Decentralised, shared-nothing**
Data storage	Multiple nodes access the shared data set	Each node accesses its private memory and disk(s)
Maintenance overhead	Dedicated directories required	Ongoing re-partitioning required
Data access speed	NAS or DFS access latency	Local disk access latency
Data shipping	Move metadata only	Move actual data at each node change
Load Balancing	Dynamic load balancing via dedicated directories	Fixed load balancing based on partitioning
Fault tolerance	Vulnerable to single-node failure	Robust to single-node failure
Availability	Depend on availability of dedicated components	Higher availability due to no single-node failure
Scalability	Dedicated components limit total number of nodes	Inherent scalability, but data shipping may downgrade scalability
Consistency	All nodes see one copy of data, and conflict is resolved by allowing multiple versions	Writes are propagated across nodes, and replicas are converged within a time period
Examples	Bigtable, HBase, and Hypertable	Dynamo, Cassandra, PNUTS, and Voldemort

Table 3.2: Comparing centralised, shared-storage vs. decentralised, shared-nothing

nothing architecture inherits several merits from DHTs. For one thing, DHT-based KVSs exhibit inherent scalability, which allows the KVS to scale to an extremely large number of nodes and to handle continual node arrivals, departures, and failures. For another, it is more tolerant of single-node failure, due to the decentralised manner of data distribution, and therefore, provides higher availability than the shared-storage model using dedicated components. Additionally, unlike previous DHTs that provide $O(\log N)$ lookups, most shared-nothing KVSs store the complete routing information in each peer locally to support $O(1)$ lookups (Ramasubramanian & Sirer, 2004), which is as efficient as using a dedicated directory as in shared-storage KVSs.

Nevertheless, decentralised, shared-nothing KVSs also have downsides. First, since the data is stored in each individual node, the requirement of data migration at node changes may downgrade the efficiency of system scaling. Second, the use of distributed hash functions limits the flexibility of employing more dynamic load-balancing strategies. Last,

the decentralisation model requires delicate design for node coordination when certain decisions need to be made.

3.1.2 Consistency Model

Data consistency is an important property for systems that replicate and distribute data over many nodes. Transactional consistency, which guarantees that the database is in a consistent state once a transaction is finished, has been the cornerstone of relational databases.

Transactional consistency is a strong consistency model, usually assessed by two correctness criteria. First, *serialisability* (Papadimitriou, 1979) requires that a history of committed transactions issue the same operations and receive the same responses as in a sequential history of transactions without concurrency. Second, *linearisability* (Herlihy & Wing, 1990) requires that all requests are ordered chronologically by their arrival time in the system and that all requests always see the effects of the preceding request. In other terms, all requests can be visualised as happening instantaneously at a single point in time and not during an interval of time.

However, the ACID requirements severely limit the scalability and availability of distributed systems. Eric Brewer neatly codifies this issue in his "CAP Conjecture" (Gilbert & Lynch, 2002), stated that a partition-tolerant distributed system can guarantee either consistency or availability, but not both. Relational databases often sacrifice availability to maintain strong consistency. In contrast, KVSs usually aim for higher availability and lower latency, and comprise on strong consistency.

An Overview of Consistency Properties

In a distributed storage system, there are two perspectives on consistency (Tanenbaum & Van Steen, 2007). From a developer's point of view (i.e. the client side), consistency is focused on how data updates are observed by different processes that need to share information. Table 3.3 shows a number of consistency properties (Tanenbaum & Van Steen 2007, Vogels 2009) that are important to consider from the client-side.

However, in the literature of KVSs, these client-side properties are not explicitly guaranteed. Instead, the standpoint is on the server side, wherein consistency is related to how data updates flow through the nodes that serve the corresponding replicas. Consistency

Consistency Properties	**Guarantee**	**Usage**
Monotonic read	A following access sees the data that is at least as fresh as what was seen before.	It helps as data versions become visible in chronological order.
Monotonic write	Two writes by the same process are serialised in the order that they arrive at the server.	It prevents the last write of a datum from being overwritten.
Read-your-writes	The process that commits a recent write, sees the data that is at least as fresh as what it just wrote.	It prevents the situation where a process re-issues the same write several times, because it sees a stale datum that it just wrote and (falsely) conceive that the previous write failed.
Causal	Writes that potentially have a causal relationship are seen by every process of the system in the same order.	It preserves all potential causality, which is evaluated via dependency trees or vector clocks.

Table 3.3: Consistency properties from the developer's perspective

is maintained with quorum-based protocols, where three parameters need to be defined:

- N is the number of nodes that serve the replicas of some data;
- R is the number of replicas that should be accessed in a valid read operation;
- W is the number of replicas that are required to acknowledge the write before a write completes.

When $W+R > N$, that is, the read and write sets always overlap by at least one node, strong consistency is guaranteed. Conversely, when $W + R \leq N$, the read and write sets may not overlap, leading to weaker consistency. There are situations where $W + R > N$ cannot be satisfied, described as below.

One situation is to maintain high availability during network partition, wherein some nodes in the system cannot reach other nodes, but both are reachable by clients. The network partition that has less than R (or W) replicas becomes unavailable for read (or write) operations.

The other situation is to provide high performance. As the latency of a read (or write) is dictated by the slowest of the R (or W) replicas, in order to provide lower latency, R and W are usually configured to be less than N. For example, $R = 1$ is the optimised read case, while $W = 1$ is the ideal write case. However, given the constraint that $W+R > N$, a

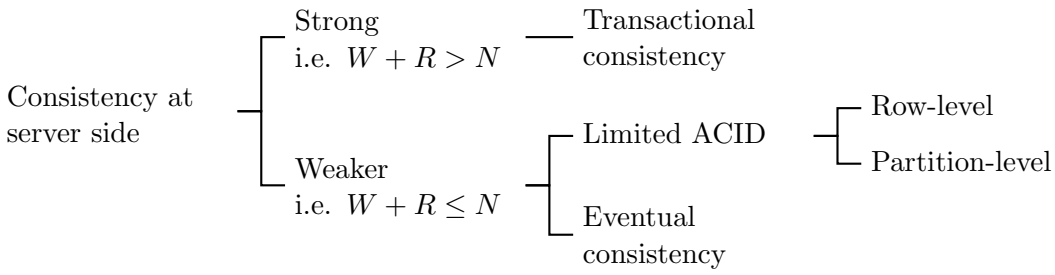


Figure 3.3: Consistency from the system's perspective

smaller R results in a greater W, and vice versa, meaning that read and write performances cannot be simultaneously optimised. Hence, for systems that aim to serve heavy read (or write) loads, the latency of write (or read) can end up to be very high.

The setback for both situations is to allow weaker consistency, where $W + R \leq N$. Since the write set is smaller than the replica set (i.e. $W < N$), a common approach is to propagate a write in a lazy manner to the remaining replicas that do not acknowledge in the first place. The time period until all replicas have updated the write is termed as the *inconsistency window* (Vogels, 2009). Hence, during this time window, it is possible that the system will return a stale value, resulting in *weak* consistency. As shown in Figure 3.3, current-state KVSs have offered various weaker consistency properties from eventual consistency to limited ACID.

Eventual Consistency

Eventual consistency (Vogels, 2009) is a specific form of weak consistency, and guarantees that if no new updates are made to a data object, subsequent accesses will return the last updated value "eventually", that is, after the inconsistency window just defined. Eventual consistency has two benefits. First, it allows temporarily disconnected replicas to remain fully available to clients, since it guarantees that updates are eventually delivered to all replicas. Second, it does not require updates to be immediately performed on all replicas, thus improving scalability.

Eventual consistency was firstly demonstrated in Amazon's Dynamo (DeCandia et al., 2007) as an appropriate trade-off for building a highly available KVS. Dynamo allows each data object to have multiple versions, differentiated using vector clocks (Lamport, 1978). Conflicting versions are reconciled using "last write wins" (Thomas, 1979). Cassandra (Apache, 2009) extends eventual consistency to provide tuneable consistency. Given

W and R defined in the previous subsection, Cassandra allows the clients to tune each value separately for any given write or read, which trades consistency against latency. As discussed, a smaller W (or R) gives lower write (or read) latency. But if both W and R are small such that $W + R < N$, then consistency guarantee becomes weak. Hence, Cassandra leaves application developers the flexibility and responsibility to control the consistency level for each read and write.

Although eventual consistency is widely adopted in KVSs (DeCandia et al. 2007, Voldemort 2009), it is criticised as "very vague in terms of concrete guarantees", and "often fulfils these guarantees (of the client-side consistency properties) for a majority of requests but does not guarantee to do so" (Bermbach & Kuhlenkamp, 2013). There are examples to back up this statement. Wada et al. (2011) observed SimpleDB (Amazon, 2008), and revealed that, when serving successive eventual consistent reads, SimpleDB provided the fresh value at the probability of only about 33% in the first read, and about 67% in the second read. SimpleDB also served stale data that violates the guarantees of read-your-writes and monotonic read consistency. Bermbach & Tai (2011) evaluated Amazon S3 (Amazon, 2006a), and also observed violation of monotonic read consistency in about 12% of all requests. Hence, an eventually consistent system do not provide safety guarantees, and can return any value within the inconsistency window.

Conditional ACID

Due to the lack of safety guarantees of eventual consistency, there are KVSs seeking conditional ACID properties over a single entity, such as a data object or a partition. These systems typically use concurrency control methods used in relational databases (Bernstein & Goodman, 1981), as well as distributed consensus algorithms (Lamport, 1998, 2001).

Transactional consistency has been guaranteed at the level of per data object (e.g., row or record). Bigtable (Chang et al., 2006) provides row-level consistency with the technique of copy-on-write, in which a copy is made for the columns that are to be updated within a row. Bigtable uses one timestamp per column to resolve conflicts between different versions. It also allows the existence of multiple versions of the same data, which are indexed by timestamp. Yahoo!'s PNUTS (Cooper et al., 2008) provides per-record timeline consistency, which guarantees that all updates of a given record (i.e. row) are applied by all the replicas in the same order. This consistency model gives the same ACID guarantees

as a transaction with a single write operation in it. PNUTS also implemented a whole range of API calls, which allow single-row transactions on a given version without any locks.

Recent research works focus on partition-level transactional consistency. ElasTraS (Das et al. 2009, 2013) uses a statically partitioned setup to limit the transactions of an application into single partitions. Then, ElasTraS leverages Chubby locking service (Burrows, 2006) that uses the Paxos consensus algorithm (Lamport, 1998), to achieve consistency at the level of partition replica (Chandra et al., 2007). Google's Megastore (Baker et al., 2011) replicates each partition of the data store separately, so as to provide fully serialisable ACID semantics within partitions. This partitioning allows Megastore to synchronously replicate each write across a wide area network within reasonable latency, which supports interactive applications. Still, Megastore provides only limited consistency guarantees across partitions. Additionally, Google's Spanner (Corbett et al., 2012), the successor to Bigtable, supports externally-consistent (Gifford, 1981) reads and writes at global scale, by sharding data across many sets of Paxos (Lamport, 1998) state machines for two-phase commits.

3.1.3 Quality Attributes

Quality attributes are non-functional requirements that specify criteria for judging the operation of a system. This thesis studies KVSs in terms of elasticity, which requires the KVS to scale in response to workload changes. Hence, the first and foremost quality attribute to investigate is scalability. Moreover, KVSs are distributed systems designed to run on hundreds or thousands of inexpensive commodity machines. Under such scale, component failures are "the norm rather than the exception" (Ghemawat et al., 2003), and can result in an unavailable system and, worse, corrupted data. Hence, the design of KVSs also focuses on the availability of system and the durability of data.

System Scalability

Scalability is the ability of a system to maintain consistent performance under an increasing workload. In other words, a system whose performance improves after adding computing resources, proportionally to the capacity added, is said to be a *scalable* system.

The scaling of a system can be horizontal, where computing resources are added or

removed from the system in the form of nodes (e.g. commodity machines). The process of horizontal scaling is also termed as "scale out" and "scale in". In contrast, scaling can also be vertical, where more computing resources (e.g., processors and RAMs) are added to a single machine. Vertical scaling is also known as "scale up" and "scale down". Nowadays, scaling is typically horizontal for distributed data stores, because low-cost commodity machines, rather than high-end machines, have been widely used.

The way that a KVS is scaled depends on its storage model, which is described in Subsection 3.1.1. In shared-nothing KVSs (DeCandia et al., 2007, Lakshman & Malik, 2010) where the data is stored in separate disks, scalability is typically achieved by data partitioning, which splits the large tables into many small partitions that can be scattered across nodes. Scalability can be further improved if the number of partitions is sufficiently larger than the number of nodes, as the overhead of re-partitioning can be avoided when the system scales within an arbitrary number of nodes. Cassandra (Apache, 2009) from version 1.1 onwards has started using a large number of "virtual nodes", a concept proposed by (Stoica et al., 2001), to build consolidated partitions that are transferable between nodes.

In contrast, in shared-storage KVSs (Chang et al., 2006, Apache, 2010) where all the KVS nodes access the same data, the scalability of a system relies on the throughput of the underlying storage. When a DFS is in use (Ghemawat et al., 2003, Shvachko et al., 2010), the data files are usually segmented into a large number of data chunks, each being replicated and distributed across a number of nodes in the DFS. That is, shared-storage KVSs rely on the underlying storage layer to scale.

System Availability

The availability of a system is measured as the proportion of time a system is in a functioning condition. To build a highly available system, one should address the failure scenarios that may cause a downtime of the services provided by the system. To ensure high availability in distributed systems, there are two design principles that have been long followed.

The first principle is to follow the mantra of "no single source of failure". It has been adopted by P2P systems, such as Chord (Stoica et al., 2001) and Pastry (Rowstron & Druschel, 2001), where the network is symmetric (i.e., each node plays the same role).

The influence of the failure of any arbitrary node is equivalent, and thus can be minimised by supplying redundant resources. Conversely, in a centralised system where certain dedicated components play the master role, the failure of such dedicated components can cause a downtime for many services in the system. There are several remedies. Bigtable (Chang et al., 2006), which uses a single master, leverages a distributed lock service called Chubby (Burrows, 2006) to elect and ensure one active master at any time. It requires very delicate implementation of the Paxos algorithm (Lamport, 1998, Chandra et al., 2007). Instead, HBase (Apache, 2010), which derives from Bigtable and uses Zookeeper (Hunt et al., 2010) as its distributed coordination service, has added support for multiple masters, since it is challenging to guarantee that a single master will always be available.

The second principle is to loosen ACID guarantees. It has been widely acknowledged by both industry (DeCandia et al., 2007) and academia (Fox et al., 1997) that data stores providing ACID guarantees tend to have poor availability. As discussed in Subsection 3.1.2, when network partition occurs, certain partitions may not possess enough replicas to meet the quorum of a strong consistency, resulting in unavailable write or read. Therefore, in order to tolerate network partitioning, the system must give up strong consistency in exchange for high availability. The approach is to allow a smaller W and R for writes and reads, where $W + R < N$ (defined in Subsection 3.1.2). For example, Dynamo (DeCandia et al., 2007) allows other live nodes to store updates for nodes that are temporarily unavailable, with the help of a "sloppy quorum" and hinted handoff. Cassandra (Apache, 2009) and Riak (Basho Tech., 2012) that are derived from Dynamo, allow the clients to tune the values of W and R for each single operation.

Data Durability

Durability means that once an update to the data set has been committed, it will survive permanently even in the event of power loss, crashes, or errors. However, when commodity machines fail, it is not uncommon that certain machines do not come back to life after the failure is recovered. In such a scenario, node failures can also cause data corruption or data loss. There are several techniques to support the durability of data when node failures occur.

One common technique is replication. Distributed file systems (DFSs) (Ghemawat

et al., 2003, Shvachko et al., 2010), on which a shared-storage KVS is built, replicate data chunks across many nodes. Each chunk is usually associated with a version number. As an example, GFS (Ghemawat et al., 2003) uses checksumming to detect data corruption in each chunk, and relies on a master to track and discover any stale replica with an outdated version number. Shared-nothing KVSs (DeCandia et al., 2007, Lakshman & Malik, 2010) make multiple copies for each write via propagation. Each operation is associated with a timestamp, and version conflicts are reconciled typically based on "last-write-wins".

Another technique is to use journals, which record the write operations in chronological order before an operation is applied. If a data node fails before the write is persistently stored, the last write can be recovered at the system restart, where the operations recorded by the journal can be re-applied. Hence, the journal is also called the "redo log" in database systems.

In addition, KVSs can also periodically create a system image for the entire data set, and back it up to a more persistent storage. The most common system image is a database dump, usually written in the form of a list of SQL (or SQL-like) statements. Another form of system image is a snapshot, which is a complete copy of the data files that are set to "read-only" at a point in time. Compared to traditional database systems, KVSs are suitable for creating snapshots, as data files are mostly immutable, that is, unalterable once the file has been created (Chang et al., 2006, Apache, 2009).

3.1.4 Query Performance

The performance of query processing (i.e. storage and retrieval of data) is affected by several factors. First, the data model (i.e. the internal data structure of key-value pairs) decides the types of data sets the KVS can support. Next, the types of supported queries determines the applicability domain. In addition, the techniques that improve data storage and retrieval are also a key performance factor. The performance of a KVS can be assessed along several dimensions and therefore, there are multiple approaches to benchmarking that are reviewed later in this subsection.

Data Model

The abstract view of data model in KVSs is a mapping from a unique *key* to a *value* that contains the information of a data object. The *key* is typically a primitive data type

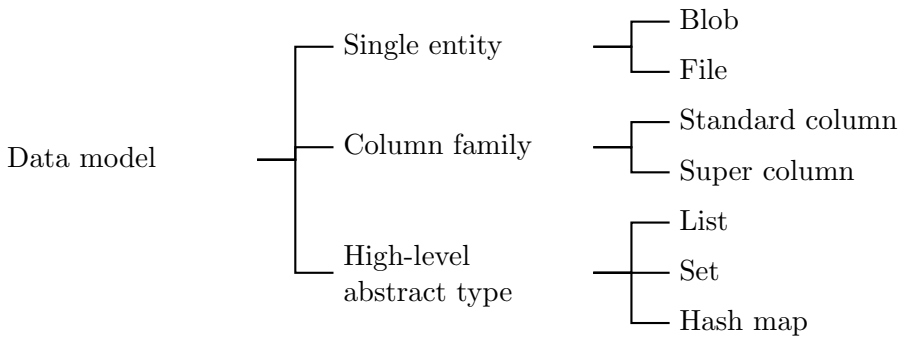


Figure 3.4: Various internal data models of a key-value pair

such as numbers, strings or byte arrays, while the *value* can use a more sophisticated data structure, depending on the type of data sets the KVS intends to support.

The most simple structure of a value is a single entity that does not have or require interpretation of its internal structure. Examples include: binary large objects (*blob*), as in Dynamo (DeCandia et al., 2007) and PNUTS (Cooper et al., 2008), and files indexed by a string, as in filesystems and traditional P2P networks.

Column family is a more flexible structure, proposed in Bigtable (Chang et al., 2006). The value is a set of columns, each having a name, a value and a timestamp. In addition, a column is called a super column if it contains another set of columns. Column family sorts the data by keys, so essentially it is a sorted, hierarchical map, which supports both structured and semi-structured data. For the sake of storage, the value is serialised and treated as uninterpreted strings. KVSs that implement column family include Cassandra (Apache, 2009), HBase (Apache, 2010), and Accumulo (Apache, 2011).

The richest data types are supported by KVSs like Redis (2009) that hosts the entire data sets in memory. The *value* of a data object can be a high-level abstract collection, such as list, sorted and unsorted set, and hash map. Redis also provides atomic support for high level operations such as intersection, union, and difference between the collections of data objects.

Query Possibilities

KVSs are designed to support the storage and retrieval of data organised as key-value pairs. Queries in KVSs have been focused on relatively simple operations such as insertion, retrieval and scanning. Based on the granularity of data that is returned, the query mechanisms can be broadly classified into three categories, namely CRUD, multivalued

access, and MapReduce support, described as below.

First of all, KVSs support four standard "CRUD" operations: Create, Read, Update, and Delete. These operations are mostly applied to a single data object, indexed by its key. Some details are worth mentioning. *Create* is to insert a data object, which is usually grouped with other data objects in memory before they are written into a sorted, immutable file. *Read* is to retrieve a data object given its key, or to scan a list of contiguous data objects given by a key range. *Update* is to modify the value of a data object, and is treated as inserting a new data object with a newer version. *Delete* is to remove a data object given its key. In most on-disk KVSs, the data object is marked as *deleted*, but the persistent data remains on disk, until the data file, which is immutable, is reconstructed in garbage collection.

The CRUD operations are only suitable for exact key match. The need for supporting more complex queries gives rise to research efforts on providing multivalued access. G-store (Das et al., 2010a) proposed a novel Key Group abstraction with the Key Grouping protocol, which provides a granule of transactional access over a set of keys that are dynamically selected. That is, G-Store uses the single key access guarantees supported by KVSs to support transactional multi key accesses. HyperDex (Escriva et al., 2012) provides a unique search primitive that enables queries on secondary attributes. It leverages the concept of hyperspace hashing, in which multi-attribute objects are deterministically mapped to coordinates in a low dimension Euclidean space. This mapping leads to efficient implementations for partially-specified secondary attribute searches and range queries. HyperDex demonstrated that partially specified objects are retrieved 12-13 times faster than Cassandra and MongoDB.

Aside from multivalued support, there are demands for using big data in KVSs to run online analytical processing (OLAP) tasks. Since MapReduce (Dean & Ghemawat, 2008) has evolved as the most widely used framework for processing large data sets, many KVSs also provide MapReduce-compatible interfaces. Hive (Thusoo et al., 2009) is a data warehouse infrastructure built on top of Hadoop. It provides SQL-like queries called HiveQL, which generates MapReduce jobs to analysis data sets stored in varied storage systems including HDFS (Shvachko et al., 2010) and HBase (Apache, 2010). DataStax Enterprise (DSE) (DataStax, 2013a) is an integration of Hadoop (Apache, 2008b) and Cassandra (Apache, 2009). In DSE, Cassandra serves data files for the input and output

of Apache Pig (Olston et al., 2008) and Hive (Thusoo et al., 2009). Other KVSs that support MapReduce include PNUTS (Cooper et al., 2008), Accumulo (Apache, 2011), Riak (Basho Tech., 2012), and DynamoDB (Amazon, 2012).

In addition, KVSs have provided varied APIs to facilitate more flexible queries other than simple retrieval and appending operations. PNUTS (Cooper et al., 2008) supports a range of API calls with varying levels of consistency guarantees. Cassandra provides a SQL-like interface called CQL (Cassandra Query Language) (DataStax, 2013b), which supports tuneable and linearisable consistency guarantees. Tables in HBase also are accessible through REST, Avro (Apache, 2012a) and Thrift (Apache, 2012b). Hence, KVSs are also termed as *NoSQL*, that is, "Not only SQL".

Data Access

KVSs are designed to provide high performance for the insertion and retrieval of data. The techniques for improving the performance of data access are associated with the storage models described in Subsection 3.1.1.

On-disk KVSs, such as Bigtable (Chang et al., 2006) and Cassandra (Apache, 2009), improve write performance by grouping the data with an in-memory data structure called *memtable*, which is flushed onto the disk at once when the memory becomes full. This strategy is suitable for workloads with majority writes, as each write is committed by an append-only journal (i.e. write-ahead log), with the value buffered in memory. The data files on disk are immutable, allowing multiple versions of certain data to be stored simultaneously on disk. In order to reduce the number of I/Os when reading a data object, each data file is built with a Bloom filter (Bloom, 1970), which is a space-efficient data structure that can be loaded into memory. In KVSs, Bloom filters are used to test the existence of a given *key* in the related data file. Bloom filters allow false-positive test results, so all the data files containing the given key will be read. As a result, multiple versions of certain data are reconciled at read.

On-flash KVSs also maintain a small amount of metadata in memory to improve performance. The difference is that, on-flash KVSs usually build an in-memory index for locating data, rather than Bloom filters that are more suitable for membership queries. For example, SkimpyStash (Debnath et al., 2011) and SILT (Lim et al., 2011) have achieved a small memory footprint (i.e. less than one byte per key). By comparison,

in-memory KVSs, such as Memcached (Danga Interactive, 2004) and Redis (2009), have already achieved extremely high performance by hosting the entire data set in RAM. However, this type of KVSs requires much higher investment in infrastructure, since RAM devices are much more expensive than hard disks and flash.

Benchmarking

Benchmarking is an effective approach to evaluating and comparing the relative performance of a system in various aspects. There have been a variety of benchmarks designed for different data storage systems, such as SPC Benchmarks (SPC, 2013) for file systems, TPC Benchmarks (Gray, 1992, TPC, 2001) for transactional evaluation, and XMark (Schmidt et al., 2002) for XML systems. Pavlo et al. (2009) focus on benchmarking batch or analytical systems such as MapReduce in Hadoop (Dean & Ghemawat, 2008, Apache, 2008b) and relational OLAP systems.

In the context of KVSs, the benchmarking focus is on scalability, availability and query performance (e.g., throughput, response time, etc.). YCSB, i.e. Yahoo! Cloud Serving Benchmark (Cooper et al., 2010), is an influential benchmark for KVSs. YCSB defines five core workloads that are usually seen in web sites, including read heavy, write heavy, read only, read latest, and short range query. It also provides choices to generate the load based on various distributions. YCSB is available as open-source, with extensible interfaces that allow developers to implement new workload packages and to assess new systems. This extensibility has made YCSB popular in evaluating KVSs. YCSB++ (Patil et al., 2011) is an extension from YCSB to support multi-tester coordination, multi-phase workloads and abstract APIs. BG (Barahmand & Ghandeharizadeh, 2013) extends both YCSB benchmarks to rate data stores using a pre-specified service level agreement (SLA), which is the essence of TPC-C benchmark (Gray, 1992). In addition, BG focuses on the processing of interactive social networking actions.

There are also research efforts on analysing the workload characteristics in large-scale data stores. Atikoglu et al. (2012) have analysed five workloads from Facebook's Memcached deployment, and reveals a number of workload patterns. For example, an examination of query performance over time clearly shows diurnal and weekly patterns, consistent with the statistics of I/O requests for Facebook's photo storage (Beaver et al., 2010). It is also observed that applications can have an extreme read/write ratio (i.e. 30:1), higher

than assumed in the literature. Other evaluations include: Netflix's experience with Cassandra (Cockroft, 2011), Facebook's Hadoop deployment (Borthakur et al., 2011), and a characterisation of workload traces in windows servers 2008 (Kavalanekar et al., 2008).

3.1.5 Key-Value Stores in Practice

Google's Bigtable

Bigtable (Chang et al. 2006, 2008) is the first generation KVS that follows the centralised, shared-storage architecture. It resembles relational parallel databases (DeWitt & Gray, 1992) in many implementation strategies. For example, Bigtable stores data in a distributed file system called GFS (Ghemawat et al., 2003), and uses a distributed lock called Chubby (Burrows, 2006) for consensus, which is similar to Oracle RAC database (Oracle, 2001). In order to improve disk writes, the data is sorted and buffered in an in-memory structure called *memtable* before being written to disk, like the updates in LSM-Tree (O'Neil et al., 1996). To improve reads, rows (i.e., data objects) are grouped and then compressed based on a client-defined locality group, similar to column-based databases (Abadi et al., 2006). Bigtable also creates Bloom filters (Bloom, 1970) for each locality group of files (called SSTables) to reduce disk accesses.

As a KVS, Bigtable differs from relational databases in its data model and consistency model. For one thing, in order to support structured data and semi-structured for a variety of applications, Bigtable uses a flexible data model called column family, which is analogous to a table in relational databases. The internal structure of column family has been discussed in Subsection 3.1.4 (page 48). For another, in order to provide highly available services, Bigtable loosens the ACID requirements, and allows the co-existence of multiple versions of the same data object (i.e. data row). Bigtable maintains row-level consistency by using a technique called *copy-on-write (COW)*, in which a new copy of the row is created when rewriting the row and the old copy persists as part of a previous snapshot.

There are many shared-storage KVSs that are modelled after Bigtable. HBase (Apache, 2010, George, 2011) is the open-source derivative, running on top of Hadoop Distributed Filesystem (HDFS) (Shvachko et al., 2010), which resembles Google's GFS (Ghemawat et al., 2003). Tables in HBase can serve as the input and output for Hadoop's implementation of MapReduce (Dean & Ghemawat, 2008), which is the standard parallel, distributed

framework for processing data on a large cluster. Hypertable (2009) is another KVS that follows Bigtable's architecture. It improves the performance of random reads over HBase, using an in-memory CellCache with adaptive memory allocation. Hypertable has been deployed by Baidu (Dong, 2009), the leading Chinese language search engine.

Amazon's Dynamo

Dynamo (DeCandia et al., 2007) is the first generation KVS that follows the decentralised, shared-nothing architecture. It focuses on providing highly available data services "at massive scale", and uses a synthesis of well known techniques to achieve scalability and availability.

To achieve high scalability, Dynamo is architected as a completely decentralised system, with the data partitioned and replicated using consistent hashing (Karger et al., 1997). This architecture closely resembles the overlay topology of Chord (Stoica et al., 2001). A gossip-based membership protocol is employed for failure detection in a totally decentralised and symmetric manner. To achieve high availability, Dynamo sacrifices strong consistency required in databases, and provides eventual consistency using a synthesis of techniques. It leverages object versioning (Lamport, 1978) as in OceanStore (Kubiatowicz et al., 2000) to resolve conflicts between replicas. Moreover, to deal with temporary node failures, it also uses a quorum-like synchronisation and hinted handoff. Dynamo has successfully demonstrated that eventual consistency can be used to design a highly available data store.

Dynamo was then followed by many shared-nothing KVSs, including Cassandra (Apache, 2009), Voldemort (2009), and Dynamo's open-source derivative called Riak (Basho Tech., 2012). Cassandra (Apache 2009, Lakshman & Malik 2010) is a hybrid of Dynamo and Bigtable. It partitions and replicates the data using Dynamo's Chord-like architecture to provide high scalability and availability, while modelling the data using column family adopted from Bigtable (Chang et al., 2006) to offer flexible, dynamic schema. Because of its great performance and flexibility, Cassandra is currently the most popular KVS, according to DB-Engines (2014a).

Memcached and Redis

Memcached (Fitzpatrick 2004, Danga Interactive 2004) is the representative of in-memory KVSs for caching. It uses the client-server architecture, wherein the servers cache the data as a unified distributed hash table based on consistent hashing (Karger et al., 1997), while the clients populate and query the table. It should be noted that Memcached is a transitory cache. That is, when the in-memory table is full, the oldest values are purged in least recently used (LRU) order. Memcached is widely used by many data-driven websites, including YouTube, Facebook, and Wikipedia. Other distributed caching KVSs are Ehcache (2009) and Hazelcast (2013). In addition, MemcacheDB (2008) is built on the Memcached protocol, although it uses Berkeley DB (Seltzer & Bostic, 1986) as its storing backend for data persistence.

In contrast, Redis (2009) is the most popular in-memory KVS that supports persistent data (DB-Engines, 2014b). It achieves data persistency in two different ways. The older version (up to 1.0) uses snapshotting, which periodically writes data from memory to disk in the format of relational database dump. The current version uses an append-only, on-disk journal to record operations processed in memory. Other persistence-enabled in-memory KVSs, such as Tarantool (2009), implement write-ahead logging and snapshotting for data persistence. Furthermore, Redis also differs from other KVSs in its data type. The *value* of a data object in Redis can be not only strings, but also abstract collections, such as lists, (sorted) sets, or mappings as in object-oriented programming language.

SkimpyStash and SILT

SkimpyStash (Debnath et al., 2011) is an on-flash KVS that is designed for high throughput server applications. It uses a hash table directory in RAM to index key-value pairs stored in a log-structure on flash. The distinguishing feature of SkimpyStash is extremely low RAM footprint at about 1 byte per key-value pair, which is more aggressive than other on-flash KVSs including FAWN (Andersen et al., 2009), BufferHash (Anand et al., 2010), ChunkStash (Debnath et al., 2010a), and FlashStore (Debnath et al., 2010b). SkimpyStash achieves low RAM footprint by moving most of the pointers (for key lookup) from RAM to flash. To improve performance (i.e. reduce flash reads per lookup), SkimpyStash uses linear chaining (Litwin, 1980) to resolve in-memory hash table collisions, and employs Bloom filters (Bloom, 1970) to avoid unnecessary lookups. Hence, the use of flash and low

RAM footprint per key allow SkimpyStash to provide 100,000 *get-set* operations/sec.

SILT (Lim et al., 2011), i.e. Small Index Large Table, is another state-of-the-art on-flash KVS. SILT is a synthesis of three basic KVSs, each with a different emphasis, i.e. on either memory-efficiency or write-friendliness. LogStore, which is write-friendly, appends individual updates (i.e. PUTs and DELETEs) to a log file on flash, with an in-memory index built based upon cuckoo hashing (Pagh & Rodler, 2004). When LogStore becomes full, it is converted to an immutable, on-flash HashStore that does not require an in-memory index. Multiple HashStores are in use at a time, until they are merged into a SortedStore, which stores key-value data in sorted key order on flash. SortedStore implements entropy-coded tries to provide an extremely compact index at 0.4 bytes per key. Overall, SILT requires only 0.7 bytes of DRAM per key and retrieves a key-value pair using 1.01 flash reads on average.

Up to this point, four categories of KVSs have been reviewed. Bigtable (Chang et al., 2006) is a centralised, shared-storage KVS that employs many techniques from database systems, but still achieves high scalability and availability by replacing the relational model with column family, and by allowing multiple versions per data object. In contrast, Dynamo (DeCandia et al., 2007) is a decentralised, shared-nothing KVS that builds on Chord (Stoica et al., 2001), and also leverages many techniques to provide highly available data services. Both KVSs have been followed by many open-source projects including Cassandra (Apache, 2009), HBase (Apache, 2010), and Voldemort (2009).

Furthermore, Memcached and Redis serve as the examples of in-memory KVSs that support extremely high performance with the price of high cost per bit in storage, while SkimpyStash and SILT are on-flash KVSs with the focus on reducing the size of metadata per key in memory. Although this thesis focuses on on-disk KVSs, there are lessons learned from these two categories of KVSs, respectively. First, how to efficiently maintain a persistent backend storage while serving data in memory. Second, how to maintain a minimised amount of indexing for the large amount of data stored on flash (or on disk).

3.2 Approaches to the Elasticity of KVSs

As discussed in Subsection 2.2.3, distributed data stores (e.g. KVSs) that are efficient in elasticity should be able to add and remove nodes at runtime, which means that the process of queries should not be interrupted by data movement during node changes.

This requires delicate management of data in a KVS for the sake of node addition and removal. This section describes the properties and techniques regarding to the algorithms for splitting the data set into partitions, the strategies for replicating the partitions among the nodes, and the approaches to moving data across the nodes.

3.2.1 Data Partitioning

Data partitioning is the division of a logical data set into distinct independent parts. Having multiple partitions facilitates the distribution of data across nodes, and the locality of group queries.

Dimensions of Data Partitioning

In the context of KVSs, partitioning is focused on the division of one table, either horizontally, or vertically, or both.

Horizontal partitioning, also known as "sharding", splits a table according to the *key* into multiple blocks for the sake of storage. That is, some tables can be very large in size, and are split and stored by multiple nodes. There are several ways to partition a table horizontally. DHT-based systems use hash functions to segment the key space into a list of buckets, termed as **virtual nodes** (Stoica et al., 2001). The mappings from virtual nodes to storage nodes can be either one-to-one as in Cassandra (Apache, 2009) (up to version 1.0), which partitions the data based on the number of nodes available. Alternatively, the mappings can be many-to-many as in Dynamo (DeCandia et al., 2007) and Voldemort (2009), where each node serves multiple *virtual nodes* (i.e. data partitions) and each virtual node is replicated to multiple nodes. Additionally, KVSs with centralised components usually group the data objects with consecutive keys into partitions, called tablets (Chang et al., 2006, Cooper et al., 2008) or directories (Corbett et al., 2012). The centralised components also shard the partitions to store a bounded volume of data, so that each partition can be served as the unit of data distribution and load balancing.

In contrast, vertical partitioning (a.k.a. "row splitting"), divides a table into multiple tables with fewer columns, with each resulting table having an independent locality. An example is the locality group in Bigtable (Chang et al., 2006), which segregates column families that are not typically accessed together into separate locality groups. In this way, each locality group provides all the columns that are required for the majority of queries,

and therefore, it avoids the operations of filtering columns so as to improve the efficiency of reads. The effectiveness of this approach has already been demonstrated in database systems (Stonebraker et al., 2005).

A table can be partitioned horizontally and vertically at the same time. For example, KVSs following the column family model can split a table into several column families (i.e. vertical partitioning). Then, each column family can be divided into multiple blocks, each storing a number of data objects (i.e. horizontal partitioning).

Data Partitioning at Node Changes

Elasticity of a KVS requires re-distribution of the data when a node is added or removed. We focus on the scheme of horizontal partitioning, because it determines the segment in which the data is located. There are various partitioning schemes in the KVSs, depending on the storage models discussed in Subsection 3.1.1.

Shared-storage KVSs, such as Bigtable (Chang et al., 2008), Spanner (Corbett et al., 2012), and HBase (Apache, 2010), do not require re-partitioning of the data when a node is added or removed. The persistent data is stored as one single image in the underlying distributed file system, such as GFS (Ghemawat et al., 2003) or HDFS (Shvachko et al., 2010). Das et al. (2011) proposed that data can be migrated by exchanging only the *metadata* of data blocks between the nodes of a database system (or a KVS), while the persistent data remains unmoved in the shared storage. A centralised controller is also used for metadata management in these schemes.

In contrast, in shared-nothing KVSs, the data has to be partitioned, because each node uses its individual storage to store a portion of the data. As discussed, the mappings between the resulting partitions (i.e. *virtual nodes*) and the nodes, can be either one-to-one or many-to-one. Accordingly, there are two approaches to data partitioning for node addition (or removal), termed as **split-move** and **virtual-node-based**, respectively. They are elaborated as follows.

Split-Move Approach

The **split-move** approach is commonly used in distributed hash tables (DHTs), and was adopted by Cassandra (Lakshman & Malik, 2010). Typically, consistent hashing (Karger et al., 1997) is used, as it introduces minimal disruption when a hash table (e.g. a key

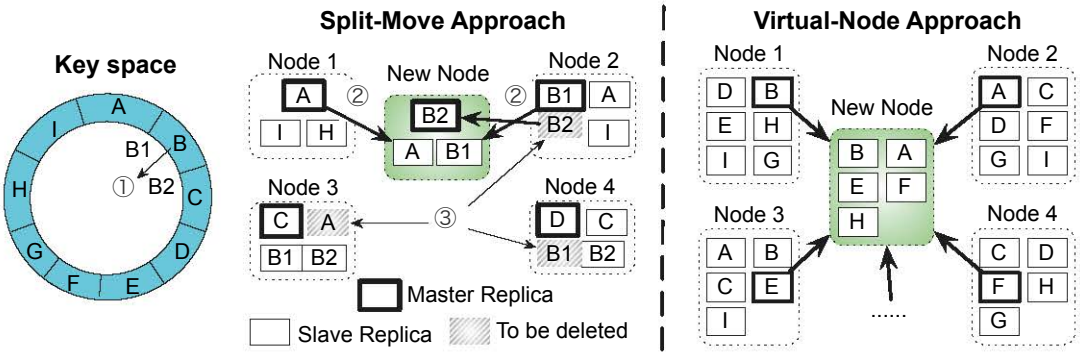


Figure 3.5: Different partitioning approaches to the addition of a KVS node

range or a partition) is resized during node changes.

As shown in Figure 3.5, the key space is split into a list of consecutive key ranges, each assigned to one node. Therefore, each node maintains one master replica for its own range, and also stores the slave replicas of several other key ranges for high availability. When a new node is added to the KVS, the key space of the data set is re-partitioned. One or several existing partitions are *split* into two sets of data (e.g. $B1$ and $B2$ as in Figure 3.5). One is retained in the existing nodes. The other set of data is *moved* out, in the form of individual key-value pairs, to the new node.

There are multiple drawbacks to this approach. One is the overhead of moving individual key-value pairs. When a partition is split, the node contributing the subset has to scan its entire dataset to prepare a list of key-value pairs for the new node, which, on receiving the data, has to reassemble the key-value pairs into files. Both scanning and reassembling are heavyweight operations.

The other drawback is that, consistent hashing aims at remapping a minimised number of keys when the number of nodes changes. As a result, only a limited number of nodes can participate in populating the new node, each undertaking relatively heavy workload. This is also true in the case of node removal. According to Amazon's experience (DeCandia et al., 2007), this approach of node addition is highly resource intensive, and is only suitable to run at a lower priority. However, low priority results in significantly slow addition of nodes, making the KVS adapt less quickly to dynamic load.

Virtual-Node-based Approach

A virtual node is a consolidated data partition that is transferable as a single unit. The idea of "virtual node" was introduced in Chord (Stoica et al., 2001) and other consistent

hashing systems, upon which KVSs such as Dynamo (DeCandia et al., 2007), Voldemort (2009), and Cassandra (Lakshman & Malik, 2010) are based. Other KVSs, such as Bigtable (Chang et al., 2008) and PNUTS (Cooper et al., 2008), use the term "tablet" instead.

This approach avoids the overhead of scanning and reassembling individual key-value pairs as in the split-move approach. In practice, the key space is over-partitioned, such that the number of virtual nodes is made much greater than the data nodes. Each data node is assigned many virtual nodes. Hence, a new node can be bootstrapped (i.e. populated with data) by multiple existing nodes, each offering one or several virtual nodes. Therefore, each participating node shares a relatively small amount of workload in bootstrapping.

However, there is a lack of efficient data partitioning schemes for completely decentralised KVSs. The current-state research efforts (DeCandia et al. 2007, Voldemort 2009) use a *simplified* partitioning strategy, wherein the key space is split into static key ranges of equal length, or hashed into buckets with equal capacity. Although this strategy avoids complex coordination amongst the peer nodes, it leads to data skew for biased key distributions. That is, a majority of keys are allocated to a minority of partitions (or buckets). Data skew results in some "giant" partitions that are difficult to migrate because of the large volume of data (Cooper et al., 2008).

One refinement is to re-hash the inserted keys using uniform hash functions, most of which, however, are not order-preserving, making the support of range queries more difficult. For those uniform order-preserving hash functions, Aberer (2011) pointed out their fundamental limitation: the key space is discrete and cannot adapt to any arbitrary application key distributions. Alternatively, PNUTS (Cooper et al., 2008) proposed to shard the tablets (i.e. partitions) into bounded sizes. However, it relies on a centralised component called tablet controller, which is not applicable to those KVSs following the decentralised architecture.

3.2.2 Data Placement

In a distributed data store, the partitions of a table (or a key space) are usually replicated to multiple nodes. The problem of data placement has been extensively studied in the literature, to meet requirements including load-balancing and data durability.

Random Placement of Replicas

In current-state KVSs (DeCandia et al. 2007, Lakshman & Malik 2010), the common approach to replica placement is to manage the data through coarse-grained structures such as buckets or virtual nodes, rather than identifying an optimal placement strategy at the granularity of single data object as proposed by Krishnan et al. (2000).

Consistent-hashing-based KVSs, such as Dynamo (DeCandia et al., 2007) and Cassandra (Lakshman & Malik, 2010), typically adopt a **random replication** strategy, wherein a hash function is used to randomly assign the replicas of groups of data objects (i.e. buckets or virtual nodes) to nodes. This strategy allows key lookups to be performed locally, in a very efficient manner (DeCandia et al., 2007). Other KVSs, such as Bigtable (Chang et al., 2006), PNUTS (Cooper et al., 2008) and Spanner (Corbett et al., 2012), rely on dedicated directory services that provide flexible mapping from virtual nodes to storage nodes. Essentially, this approach also uses random replication strategy, because both virtual nodes and storage nodes are randomly chosen, without explicit constraint on the location of data.

Although random replication merits in simplicity and efficient key lookup (DeCandia et al., 2007), it also has two disadvantages. First, due to the ignorance of the access frequencies of nodes to data, the placement may result in highly sub-optimal performance in terms of load balancing. Second, random placement of data replicas leads to higher probability of data loss when multiple nodes fail simultaneously. Cidon et al. (2012, 2013) have demonstrated that random replication is nearly guaranteed to cause a data loss event once the size of the system scales is beyond hundreds of nodes.

Data Placement For Load Balancing

There has been research on the problem of optimising data placement based on the workload for better load balancing and query performance.

Ceph (Weil et al., 2006) performs dynamic placement of individual data objects, with the aim to evenly distribute loads over all nodes. However, Anderson et al. (2005) pointed out that the problem of finding load balancing schemes with minimal re-configuration costs is just a variant of bin-packing or knapsack, which is inherently expensive and solvable offline. Instead, Everest (Narayanan et al., 2008) focused on dealing with peak loads, by off-loading the workload from overloaded storage nodes to under-utilised nodes.

Moreover, Ursa (You et al., 2011) formulates the problem of workload-aware replica placement as an integer linear programming (ILP) problem, with the goals to eliminate hot-spots while minimising the data movement cost. Yet, Ursa (You et al., 2011) relies on centralised components to compute the placement and to maintain a location map, which is not applicable to decentralised KVSs.

In contrast, AUTOPLACER (Paiva et al., 2013) uses a decentralised algorithm that optimises the placement of the top-k most frequent data objects in a self-tuning manner. It combines the usage of consistent hashing with a probabilistic mapping strategy that operates at the granularity of the single data object. AUTOPLACER relies on Bloom filters (Bloom, 1970) and decision-tree classifiers (Domingos & Hulten, 2000), to achieve a highly efficient re-allocation of a very large number of hotspot data objects.

Additionally, others research efforts (Laoutaris et al., 2006, Zaman & Grosu, 2011) also proposed distributed algorithms for the problem of replica placement. However, these efforts only consider the placement of read-only replicas.

Data Placement for Data Durability

Replica placement and its impact on data durability has been studied extensively in the past. In order to achieve high data durability, peer-to-peer system evaluations (Weatherspoon & Kubiatowicz 2002, Rodrigues & Liskov 2005) have considered replication (Lee & Thekkath 1996, Dabek et al. 2001, Ghemawat et al. 2003) versus coding techniques (Patterson et al., 1988), and concluded that replication provides better robustness to survive the high rate of failures in distributed infrastructures.

In the context of DHT systems, Chun et al. (2006) have identified that randomly replicating data across a large set of nodes increases data loss probability under simultaneous failures. They investigated the effect of different sizes of node sets for replication using their DHT system called Carbonite, which creates and keeps track of additional replicas to handle small-scale failure events at a low cost. Yu et al. (2006) also analysed the replication strategies used in DHTs (Stoica et al. 2001, Rowstron & Druschel 2001, Ratnasamy et al. 2001), and proposed to improve availability by constraining the placement of replicas within the DHT-structured called “Group”. Similarly, Glacier (Haeberlen et al., 2005) constrains each replica (of a partition) to be placed at equal distances in the keys’ hash space. Glacier also trades efficiency in storage utilisation for durability, by maintain-

ing massive redundancy of data to provide highly durable data. However, this approach is expensive in terms of maintaining data consistency over a large number of replicas.

Cidon et al. (2013) also proposed to confine the placement of replicas into certain predefined sets of nodes, called "copysets". Hence, the failure of a copyset of nodes is equivalent to the loss of that unit. The copyset-based replication schemes focus on minimising the number of copysets in the system to provide high data durability. They compared this copyset-based strategy against random replication, and proved that the copyset-based strategy could significantly reduce the probability of data loss when a non-negligible percentage of nodes (e.g. 1%) fail simultaneously.

However, existing implementations of copyset-based replication (Cidon et al. 2013, 2012) rely on random permutation to create copysets. For each node addition, new copysets are formed without altering any existing copyset, while no affected copysets are dismissed at each node removal. Instead, an existing node is randomly selected to replace the removed node. This approach ends up in increasing number of copysets if nodes join or leave dynamically, and therefore, is not applicable to KVSs that require elasticity in response to workload changes.

There are other research efforts on data durability. Large companies, as in the case of Facebook (Hamilton, 2008) and Google (Ford et al., 2010), consider geo-replication as an effective technique to prevent data loss under large scale concurrent node failures. However, this approach is not feasible for storage providers that operate within one region. Disaster recovery systems (Chang et al. 2002, Patterson et al. 2002) use replication and mirroring to increase durability. These systems focus on the cost of propagating updates on data files that are alterable, which is not applicable to KVSs that store data in immutable files (DeCandia et al. 2007, Lakshman & Malik 2010).

In addition, the relation between data durability and the number of possible replicas has also been discussed previously (Saito et al. 2004, Van Renesse & Schneider 2004). The trade-off here is between reducing the probability of losing a data object during a simultaneous failure (by limiting the number of replicas) and improving the robustness to tolerate a higher average failure rate (by increasing the number of replicas). However, none of these have discussed the trade-offs between data durability and scalability.

3.2.3 Data Migration

The elasticity of KVSs requires the nodes to be added or removed at runtime. To efficiently deal with node addition and removal, these arises the need for online data migration, the focus of which is on maintaining the ability to execute queries while data movement is in progress, which is termed as *live migration* (Das et al., 2010b).

Shared-storage KVSs, such as Bigtable (Chang et al., 2006) that runs atop GFS (Ghemawat et al., 2003) do not require actual data migration. Instead, each new KVS node only needs to acquire small amounts of metadata or in-memory cache for a warm startup. For example, Albatross (Das et al., 2011), which targets at the shared-storage architecture, provides a lightweight migration solution that simply hands over the identifiers (i.e. metadata) of data blocks across nodes. Albatross also uses a technique called Iterative Copy (Das et al., 2010b) to replicate the caches and metadata (i.e. file pointers) from the source node to the destination, with no aborted transactions, such that the destination node starts with a hot cache.

In contrast, data migration is inevitable in shared-nothing KVSs when node changes occur. There has been prior research on maintaining data availability during node addition or removal. Relational Cloud (Curino et al., 2010a) proposed various migration strategies, including splitting a partition into smaller partitions for incremental migration, and prefetching data to prepare warm stand-by copies. Zephyr (Elmore et al., 2011) minimises service interruption with a synchronised *dual mode*, where the source and destination nodes are both enabled to execute transactions. However, Zephyr is intended for migrating small amount of data. In practise, the process of data migration can be extremely slow. As reported by Amazon, Dynamo "has taken almost a day to complete" data migration for adding a new node (DeCandia et al., 2007), because the throughput of data transfer is throttled to avoid introducing large amount of workloads to the source nodes. There is a lack of strategies that improves migration performance while maintaining service availability.

Slacker (Barker et al., 2012) is the state-of-the-art live migration solution, and is designed for CloudDB (Hacigumus et al., 2010). Slacker makes use of the "slack" (i.e. idle) resources for migration without seriously impacting workloads that are already present on the nodes. It is performed in three steps: i) *snapshot* is prepared and transferred from the source node, which continues to serve queries during this step; ii) the *delta* is calculated

by examining the query log of the source node; iii) a freeze-and-*handover* is performed to copy the delta to the destination node. This snapshot-delta-handover approach is similar to the VM migration technique employed by Clark et al. (2005), who also discussed some extreme cases such as workloads with very high write turnover.

Additionally, migration can also be performed implicitly by replication. Schism (Curino et al., 2010b) uses a workload-driven, graph-based algorithm to decide the number of partitions that each tuple should be replicated. The decision is made by trading off the costs of distributed updates against distributed transactions. EcStore (Vo et al., 2010) extends the BATON tree (Jagadish et al., 2005) to provide a two-tier replication mechanism, where ecStore creates secondary and slave replicas in addition to the primary copy. The secondary replicas are maintained to provide data availability, while the slave replicas are created for popular data objects to improve load balancing. Other research (Savinov & Daudjee, 2010) attempts to provision backup replicas when the master server is heavily loaded or even fails, wherein the replicas are created via database dump, which contains a definition of the table structure and the data from a database, usually written in the form of a list of SQL (or SQL-like) statements.

3.3 Discussion and Summary

This chapter has provided a thorough survey of the design choices in KVSs. This survey is essential to carry forward the lessons learned from the rich literature, presented based on four aspects. First, the design choices in system architecture, such as storage model and coordination model, separates a KVS from one category to another, such as centralised, shared-storage KVSs and decentralised, shared-nothing KVSs. Second, the consistency model is a dominant factor to system availability in light of "CAP Conjecture" (Gilbert & Lynch, 2002), i.e., weaker consistency model results in better availability. Moreover, quality attributes, such as scalability, availability, and data durability, specify criteria to assess the quality of system operations. Last, the variety and performance of query directly affect user satisfaction. Multiple approaches to benchmarking were also reviewed for the assessment of a KVS's performance in various aspects. Given this survey, this chapter has also studied a number of state-of-the-art KVSs that have significant influence to the development of KVSs.

Based on the background of KVSs introduced, this chapter continued to examine the

schemes of data management in KVSs in terms of efficient node addition and removal for the sake of elasticity. To begin with, this chapter has identified the need for data partitioning in shared-nothing KVSs, and then compared two common strategies to data partitioning, termed as *split-move* and *virtual-node-based*. It has concluded that *split-move* is inefficient in data migration, while *virtual-node-based* approaches rely on either hash functions for static partitioning, which leads to the issue of data skew, or centralised components for distributed coordination, which is not applicable to decentralised KVSs. Hence, there is a lack of decentralised scheme of data partitioning for shared-nothing KVSs to efficiently add or remove nodes.

Moreover, this chapter has reviewed the strategies of data placement in current-state KVSs, most of which use random replication that randomly assign replicas of data to nodes. As discussed, this strategy leads to two issues: i) sub-optimal performance in load balancing due to the ignorance of the access frequencies of nodes to data; and ii) higher probability of data loss when multiple nodes fail simultaneously. Consequently, there arose research, including Ursa (You et al., 2011) and AUTOPLACER (Paiva et al., 2013) that address the problem of workload-aware data placement.

The problem of data durability at multiple node failures was tackled by constraining the placement of data replicas at particular nodes (Haeberlen et al. 2005, Chun et al. 2006, Yu et al. 2006). Cidon et al. (2013) proposed the concept of "copyset" to confine the placement of replicas into certain predefined sets of nodes. This copyset-based placement strategy minimises the probability of data loss at node failures. However, the formation of copysets is calculated offline, and is not adaptive to dynamic node changes. Hence, there is a lack of data placement strategy that minimises the data loss at multiple node failures, while allowing dynamic node addition and removal at runtime.

Last but not least, this chapter has also reviewed the approaches to moving data across nodes without affecting the execution of queries, termed as *live migration* (Das et al., 2010b). The current-state solution is Slacker (Barker et al., 2012), which effectively makes use of the "slack" (i.e. idle) resources for migration without seriously impacting workloads that are already present on the nodes.

Now that the research gaps in data partitioning and placement have been identified, the rest of this thesis will elaborate a set of data management schemes for the efficient elasticity of KVSs.

Chapter 4

Data Distribution for Efficient Elasticity of KVSs

Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time.

– Thomas A. Edison

This chapter presents the design of a data distribution middleware that improves the efficiency of elasticity for decentralised shared-nothing KVSs, while the implementation and evaluation of this middleware will be presented in the next chapter. This chapter starts with an analysis of the challenges of efficient elasticity in the decentralised, shared-nothing architecture. Then, it presents the detailed design of a set of data distribution schemes. Finally, this chapter concludes with a general discussion on the contributions of this middleware.

4.1 The Elasticity Challenge

Distributed KVSs (Chang et al., 2006, DeCandia et al., 2007, Cooper et al., 2008) have become a standard component for many web services and applications due to their inherent scalability, reliability and data availability, even in the face of hardware failures. While KVSs have been mostly used in data centres, many enterprises are now adopting them for use on servers leased from the Infrastructure-as-a-Service (IaaS) Cloud.

As discussed in Section 2.1, computing resources from the IaaS Cloud are typically in the form of virtual machines (VMs), and can be provisioned or de-provisioned anytime

on-demand. To deal with increasing workload, new VMs are acquired to improve the system's capacity. Since IaaS providers normally follow the "pay-as-you-go" pricing model, redundant VMs can be shut down in the face of declining demand to save on economic costs. In the remaining of this thesis, the process of incorporating a new empty VM as a member of KVS is termed as node *bootstrapping*. In contrast, the process of eliminating an existing member with redundant data off the KVS is called node *decommissioning*. In this regard, efficient *elasticity* requires a KVS to bootstrap or decommission a node quickly and frictionlessly, i.e. without affecting online query processing.

The storage model of a KVS determines the performance of data movement during node bootstrapping and decommissioning. In shared storage KVSs, the persistent data is stored in the underlying storage, such as networked attached storage (NAS) or distributed file system (DFS), which is accessed by all the nodes. The data can be migrated between nodes without actual data transfer. An example is Albatross (Das et al., 2011), which exchanges the metadata (e.g., identifiers or ownership) of data blocks located in the shared storage to achieve lightweight data migration.

In contrast, shared-nothing KVSs consist of distributed nodes, each with their own separate storage, coordinated as a distributed hash table (DHT). When a new node joins (or leaves) the KVS, it has to obtain (or give away) data over its peer nodes. This process is usually throttled, or run in a low-priority thread, so as to avoid affecting the online query processing by consuming network bandwidth and computational processing capacity. However, as revealed in Amazon's experience (DeCandia et al., 2007), this strategy significantly slows the bootstrapping process, especially when the query workload is high.

It is non-trivial to efficiently bootstrap or decommission a node for shared-nothing KVSs. The goal of efficiency is three-fold. First, the negative impact of data movement against online query processing should be minimised. Second, data consistency and availability should be maintained during bootstrapping and decommissioning. Third, after node changes, the load, in terms of both data volume and query workload that each node undertakes, should be re-balanced as quickly as possible. KVSs that handle node changes with such requirements are *efficiently elastic*.

The challenge of efficient elasticity in the shared-nothing KVSs lies in the data distribution strategies when a node is bootstrapped or decommissioned. Specifically, it requires

a mechanism that *partitions* the key space of a data set and then *reallocates* the partition replicas (i.e., via data migration) during node arrival or departure. Moreover, most shared-nothing KVSs (DeCandia et al., 2007, Lakshman & Malik, 2010) are essentially DHTs, deployed in a completely decentralised architecture (i.e., a P2P network). There is a need for decentralised coordination between the nodes to execute data distribution. However, as discussed in Subsection 3.2.1, the current-state data partitioning strategies, i.e. *virtual-node-based*, rely on either hash functions for static partitioning, which leads to the issue of data skew, or centralised components for distributed coordination, which is not applicable to decentralised KVSs.

This chapter describes the design of a middleware layer that provides a decentralised scheme of data partitioning and reallocation to improve the efficiency of node changes in shared-nothing KVSs. The main contribution of this work is a decentralised automated partitioning scheme that consolidates each partition of data into single transferable replicas. This scheme is extended from the *virtual-node-based* partitioning strategies. It eliminates the overhead of migrating individual key-value pairs, and overcomes the issue of data skew introduced by static partitioning in the *virtual-node-based* approach. Through automated partitioning, the data volume of each partition replica is confined into a bounded range.

This chapter also discusses the related placement and migration schemes for the consolidated partition replicas. The placement algorithm evenly reallocates the replicas when a node is bootstrapped or decommissioned, with the objectives of: i) rebalancing the data volume and workload each node undertakes; ii) maintaining high data availability; and iii) minimising data movement at startup for quick bootstrapping. The data migration leverages a token ownership mechanism to allow online query processing with eventual consistency guarantees, while a partition replica is being migrated from the source node to the destination.

4.2 Design of An Elasticity Middleware

The targeted system follows the typical decentralised shared-nothing architecture. As depicted in Figure 4.1, each node (denoted as n_i) of a KVS runs in one single virtual machine (VM), provisioned from IaaS providers such as Amazon EC2 (Amazon, 2006b). Each VM is attached to an individual persistent storage on which the data assigned to the

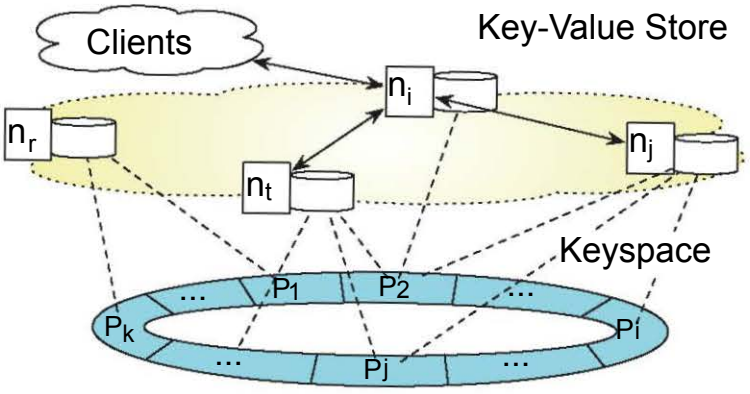


Figure 4.1: The decentralised shared-nothing architecture of Key-Value Stores

node is stored. The key space of a data set is split, or hashed, into multiple partitions. Each partition, denoted as P_i, is replicated to multiple nodes for high availability, and each node serves many partitions for load balancing purposes.

The KVS processes queries similar to DHTs such as Chord (Stoica et al., 2001) and Pastry (Rowstron & Druschel, 2001). Clients can connect to any node and execute CRUD (create, read, update and delete) operations given the key(s). We term the node to which a client is connected as the master node for the operation (i.e., n_i in Figure 4.1). The master node consults with the routing information to prepare a list of nodes that serve the key. The query is applied to those nodes via remote procedure call (RPC) and the result is returned to the client via the master node. Unlike DHTs that use multiple-hop routing, each node of the KVS maintains enough routing information locally so as to route a request to the appropriate node directly, i.e. in *0-hop* (Gupta et al., 2003b).

This section discusses the design of a middleware layer that sits between the key space of a data set and the storage of nodes. The design assumes the presence of a number of techniques that are already implemented in the KVS. First, a membership and failure detection protocol, such as gossip-based messaging (Van Renesse et al., 1998). Second, a sloppy quorum approach, like hinted handoff (DeCandia et al., 2007), to tolerate temporary node or network failure. Last but not least, data objects are assigned with timestamps, and a timestamp-based reconciliation is used to ensure eventual consistency.

The following of this section describes the synthesis of an automated partitioning algorithm, a decentralised coordination scheme, a replica placement strategy for dynamic node changes, and a data migration approach that maintains online query and data consistency, which together improve the efficiency of elasticity for decentralised, shared-nothing KVSs. The notational conventions used throughout the chapter are summarised

Table 4.1: Notational Conventions

Notation	Description
N	The number of nodes in a KVS
Q	The number of partitions of a key space
n_i	The i^{th} node in a KVS
P_i	The i^{th} partition of a keyspace
T_i	The i^{th} token that defines the boundary of P_i and P_{i+1}
ν_i	The total number of replicas of P_i
Θ_{max}	The upper-bound size of a partition
Θ_{min}	The lower-bound size of a partition
K	The replication number
L	The consistency level
$h_{i,t}$	The *local* hit count of P_i between time $t-1$ and t
$H_{i,t}$	The moving average of the *local* hit count of P_i at time t
W_i	The workload of the node n_i in certain period
R_i	The total number of replicas that the node n_i has
i, j, k	Non-negative integers
$\{x_i\}_{i=1}^{n}$	The set of n elements, wherein the argument x could be any variable
σ_x	The standard deviation of $\{x_i\}_{i=1}^{n}$
$\overline{x}$	The average value of $\{x_i\}_{i=1}^{n}$

in Table 6.1.

4.2.1 Automated Data Partitioning

The aim of automated partitioning is to confine the actual volume of data in each partition into a bounded range. The benefit is three-fold. First, partitioning is adapted to the change of data volume, thus eliminating the concern of data skew in the virtual-node-based approach. Second, each partition replica can be consolidated as a single storage unit, which can be easily moved across nodes without extra preparation of data, such as scanning or reassembling. Third, the task of load-balancing becomes easier, because every partition replica contains similar volume of data, which means balancing the number of replicas in each node can result in a balanced distribution of data.

This subsection describes the partitioning strategy in the view of the key space, and the next subsection presents the coordination over a cluster of decentralised nodes to execute an automated partitioning task in a decentralised manner.

A Varient of Consistent Hashing

The partitioning algorithm builds on consistent hashing (Karger et al., 1997), in which the largest hash value is wrapped around to the smallest to form a fixed circular space

or "ring". As shown in Figure 4.2a, when a data table is created (or declared), a sorted list of tokens, $\{T_i : 0 < i \leq Q\}$, are generated to segment the key space of the data into Q consecutive equal-size key ranges. Note that in consistent hashing, $T_0 = T_Q$, and Q is configurable by the KVS administrators. Each key range defines one partition of data. Therefore, each partition P_i can be associated with the token T_i, which defines the upper bound of P_i. The lower bound is determined by the predecessor T_{i-1}. To lookup a key, simply walk the ring clockwise to find the first token that is larger than the key. Thus, the key locates in the partition represented by the found token.

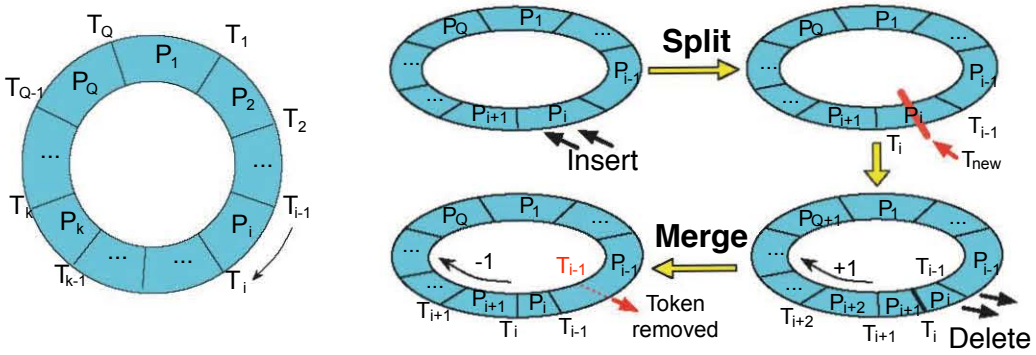
This variant of consistent hashing was adopted by Dynamo (DeCandia et al., 2007), and has been widely followed by Project Voldemort (2009), Riak (Basho Tech., 2012), and Cassandra (Apache, 2009, version 1.2 onwards). As discussed, it is a static partitioning strategy, which is prone to data skew issues when the key distribution of the data set is biased. To overcome this limitation, this thesis proposes an automated partitioning algorithm extended from this variant.

Definitions of Split and Merge

Let $Size(P_i)$ be the data volume of the partition P_i, termed as the partition size. The maximum size Θ_{max} and the minimum size Θ_{min} are constant, and are defined as in Equation 4.1 before the KVS starts up. Hence, a partition P_i is *split*, when $Size(P_i)$ exceeds Θ_{max} as a result of data insertion. At least one resulting partition after the split has a partition size larger than $\Theta_{max}/2$. Conversely, two adjacent partitions (e.g., P_{i-1} and P_i) are *merged*, when their total data volume falls below Θ_{min} due to data deletion. Therefore, the size of the merged partition is less than Θ_{min}.

$$\begin{cases} \forall i \in [1, Q], Size(P_i) \leq \Theta_{max} \\ \forall i \in [1, Q], Size(P_{i-1}) + Size(P_i) \geq \Theta_{min} \end{cases} \tag{4.1}$$

To avoid oscillation between split and merge, it is required that $\Theta_{max} > \Theta_{min}$. In practice, we set $\Theta_{max} \geq 2\Theta_{min}$. Therefore, when a partition is equally split (i.e., original $Size > \Theta_{max} \geq 2\Theta_{min}$), each sub-partition size is no less than Θ_{min}, which does not trigger a merge even if the size of the adjacent partition is zero. Also, when two partitions are merged, the resulting partition size is approximately $\Theta_{max}/2$, which is only halfway down to trigger a split.



(a) Consistent hashing (b) Split or merge partitions based on data volume

Figure 4.2: The automated data partitioning algorithm

Given Θ_{max} and Θ_{min}, we can also provide an estimation of the actual data volume of a partition on average. It is assumed that data objects (i.e., key-value pairs) are randomly inserted and deleted across the key space and over a long run. We consider the total size of any two adjacent partitions. According to Equation 4.1, the maximum total size (of two partitions) can be $2\Theta_{max}$, while the minimum total size can be Θ_{min}. Since the data is randomly scattered, the average total size of two adjacent partitions is close to the median of the maximum and the minimum, which is $(2\Theta_{max} + \Theta_{min})/2$. Hence, it can be estimated that the average data volume of a partition, denoted as $\overline{Size(P)}$, is given by Equation 4.2.

$$\overline{Size(P)} = \Theta_{max}/2 + \Theta_{min}/4 \tag{4.2}$$

This estimation helps the KVS administrators to decide the values of Θ_{max} and Θ_{min}. For example, let $\Theta_{max} = 2\Theta_{min}$. Then, the average partition size is $0.625\Theta_{max}$, that is, 62.5% of the upper bound.

Split and Merge Operations

Figure 4.2b illustrates the management of key ranges during partition split or merge. When the partition P_i is *split*, a new token T_{new} is chosen between the key range of T_{i-1} and T_i, such that the resulting sub-ranges $(T_{i-1}, T_{new}]$ and $(T_{new}, T_i]$ contain roughly equal volumes of data. Since the tokens form a sorted list, when T_{new} is inserted into the ring, T_{new} becomes the new T_i. The original T_i becomes T_{i+1}, and T_{i+1} becomes T_{i+2}, and so forth. Although the insertion of T_{new} increases the ordinals of its following tokens, it only changes the key range of P_i, which is split into the new P_i and P_{i+1}.

To merge two adjacent partitions P_i and P_{i+1}, the token T_i that sets the boundary of these two partitions is removed. Thus, the merged key range is $(T_{i-1}, T_{i+1}]$. Similarly, when T_i is removed, the original T_{i+1} becomes the new T_i, and the ordinals of the following tokens move forwards. Thus, P_i and P_{i+1} are merged as the new P_i.

By comparison, the operation of split is straightforward since it is associated with only one partition. However, the operation of merge involves with two partitions, and thus requires additional considerations. On one hand, since a partition with small volume of data is easy to migrate over the nodes, there is less harm in retaining "sparse" partitions that contain small amount of data, rather than merging them aggressively, which introduces extra workload. On the other hand, keeping the sparse partitions downgrades load-balancing, as these partitions are more likely to attract fewer workloads. In addition, maintaining a large number of partitions also increases the overheads of metadata management.

Hence, we attempt to merge partitions whenever applicable, except for two scenarios. First, two adjacent partitions will not be merged, if they are not stored on the same *set* of nodes. Note that each partition is replicated to a *set* of nodes. A merge will not be executed if there exists at least one node that stores only one of the two adjacent partitions. In most cases, it is not necessary to move the two partitions onto the same set of nodes for the sake of merging partitions. However, it does no harm to move the partitions if both the source and destination nodes have sufficient idle resources. This idea is similar to Slacker (Barker et al., 2012), which proposes "slack migration". Second, we try to maintain a minimum number of partitions in each key space. If the actual number of partitions is no greater than the predefined value Q, then the merge operation will not be triggered. This is to restrict excessive merging of partitions, especially when a table is recently created while the data is not yet populated.

Rebuilding Replicas for Automated Partitioning

The challenge of automated partitioning, either split or merge, lies in the consolidation of each partition replica as a single transferable unit. It requires that, data objects (i.e. key-value pairs) belonging to different partitions, should be stored in separate data files. A consolidated replica can be transferred in the form of a list of data files, rather than key-value pairs. Compared to KVSs that do not specifically use separate files for different

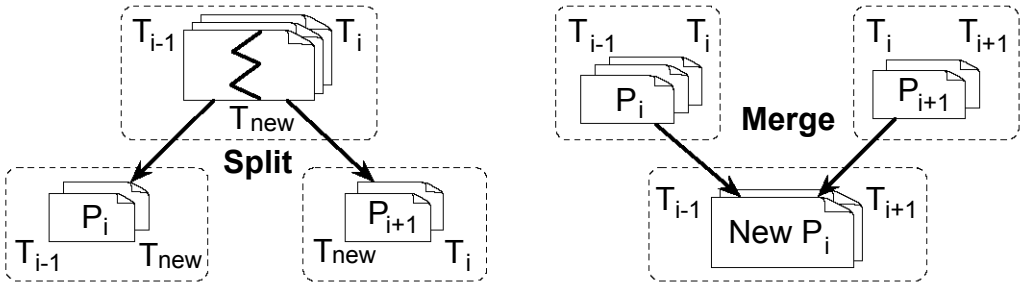


Figure 4.3: Rebuild replicas during automated partitioning

partitions, this consolidation reduces the overheads of data migration in two ways. First, it saves the computation of serialising (or marshalling) key-value pairs for the sake of transmission. Second, it eliminates the scan for every desired key-value pair to be reallocated, and the reassembling into data files at the destination.

Figure 4.3 illustrates how to rebuild a consolidated replica when a partition is split or merged. When splitting a partition P_i with a token T_{new}, the data files of a P_i replica is rewritten into two groups of files: one stores the key-value pairs with the keys ranging between T_{i-1} and T_{new}, while the other stores the data between T_{new} and T_{i+1}. In contrast, when merging P_i and P_{i+1}, the data files of these two partitions are rewritten into one group of files, which stores key-value pairs ranging between T_{i-1} and T_{i+1}.

Rebuilding replicas of the affected partitions is the key operation in automated partitioning. It takes off the overheads in data migration during node bootstrapping and decommissioning, by preparing the partition replicas as transferable units in advance. However, since each partition has multiple replicas in different nodes, it requires a decentralised coordination scheme to accomplish the task of rebuilding replicas. This coordination scheme is presented in the following subsection.

4.2.2 Decentralised Coordination

Coordination of Data Partitioning

Since each partition is replicated to multiple nodes, the operation of rebuilding each local replica is executed by different nodes asynchronously. Thus, coordination is required to ensure the consistency of the key space and persistent data across the nodes that participate in partitioning. As shown in Figure 4.4, data partitioning in a distributed KVS is coordinated in four steps. In the following discussion, the partitions that are to be split or merged, are termed as the *targeted partitions*, while the nodes that host the

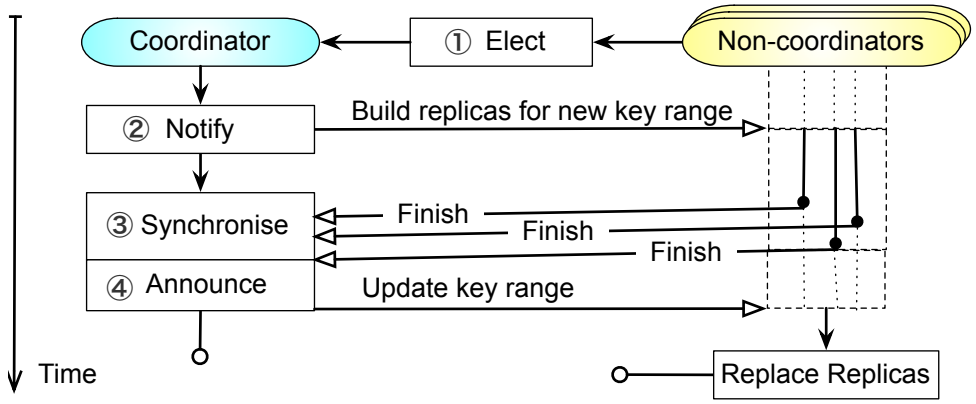


Figure 4.4: Automated partitioning with election-based coordination

targeted partitions are termed as the *participating nodes*.

Step 1: Election.

When the data volume of certain partition replicas reaches the boundary (Θ_{max} or Θ_{min}), the node that serves the targeted partitions initiates a partitioning operation. This operation can be initiated by multiple participating nodes simultaneously. Hence, a coordinator node should be nominated to supervise the whole operation.

The election of a coordinator relies on the Chubby implementation (Burrows, 2006), which is extensively used by Bigtable (Chang et al., 2006) for a variety of distributed tasks. According to Chubby, the coordinator must obtain votes from a majority of the participating nodes. The votes are also attached with the promises that those participating nodes will not elect a different coordinator for a time interval known as the *master lease*, which is periodically renewed. In our design, the node that initiates the partitioning retrieves the complete list of nodes that serve the partition. The list is sorted by certain criteria, and the node on top is voted as the *coordinator* (for this single operation only). The other participating nodes also vote for the node on top of the list. Thus, the node that receives a majority of votes becomes the coordinator. Note that the coordinator is also a participating node that rebuilds its replica.

There are two challenges with this election mechanism. First, some participating nodes may use an outdated partition-node mapping, and vote for different nodes. The remedy is to force the nodes to "gossip" their lists of nodes serving the targeted partitions before the election. Each node updates the gossip message, by answering whether it owns the targeted partitions or not. Thus, the information of partition-node mapping is updated via gossiping. Second, in some KVSs, each partition is replicated to only two or three nodes.

A consensus cannot be reached if one or two of the participating nodes fail to respond. To address the issue that a consensus cannot be reached among the participating nodes, a public vote is initiated instead, where all the living nodes of the KVS are invited to vote in the election.

Step 2: Notification.

The elected coordinator makes all the decisions for this single partitioning operation. In the case of a *split*, the coordinator calculates the splitting token T_{new} based on its own the local replicas of the targeted partition. This token T_{new} will be used by all the participating nodes. In the case of merge, the coordinator examines whether the extra prerequisites for merging are satisfied. For example, the two targeted partitions should be located on the same list of nodes. And if they are not, the coordinator decides whether a reallocation of the targeted partitions can be performed between any idle nodes available. In our design, the coordinator identifies an idle node based on a threshold of the cpu utilisation, which is pre-defined by system administrators (e.g., cpu usage below 30%).

Once the prerequisites are met, the coordinator notifies that a split or a merge should be launched. Then, all the participating nodes start to rebuild their own replica after they receive the notification. On the other hand, if the prerequisites (of a merge operation) are not met, the coordinator will cancel this operation. To prevent the participating nodes from repeatedly initiating the same merge operation that is to be canceled, the coordinator notifies that this operation should not be submitted within a long time period.

Step 3: Synchronisation.

The operation of *rebuilding replicas* is executed within the participating nodes. Depending on the size of replicas and the workload each node undertakes, the time taken to rebuild the replicas can be varied. This operation is completed asynchronously by different nodes. When a node finishes, it notifies the coordinator and then waits for further announcement. The coordinator synchronises this operation until all the participating nodes have finished.

During this operation, each participating node creates the internal replicas for the new partitions after split or merged, but still uses the original partitions for query processing. From the perspective of the non-participating nodes, the key space remains unchanged, unless the coordinator makes relevant announcement.

Step 4: Announcement.

Once the coordinator has received the acknowledgement of *Finish* from all the participating nodes, it notifies the participating nodes to replace the old replicas with the newly-built replicas. The participating nodes update the internal file pointers and acknowledge the replacement immediately. Finally, the coordinator announces globally that the key range of the affected partition should be updated to the new range. On receiving this final announcement, every node in the KVS updates the query routing information. At this point, the partitioning operation is completed.

Failover During Data Partitioning

Based on the four-step coordination described, we discuss how to handle node failures during a partitioning operation. This failover scheme aim to achieve two goals. First, if only one participating node fails during the process, the partitioning can proceed and succeed regardless of whether the failed node can resurrect or not. Second, if more than one participating node fail, the partitioning operation can be aborted and rolled back without data loss.

The failure detection in our system is gossip-based (Van Renesse et al., 1998). The first challenge is the failure of failure detection. We assume detection error exists, since a failure detector is not always completely accurate. There are two types of detection errors. A *false-negative* detection error, due to the delay in detection, considers a node as still alive, while the node is actually dead. In our design, the communication between a coordinator and a non-coordinator follows the typical three-way handshaking policy as in a TCP connection. Hence, this type of error can be easily detected, because each message requires an acknowledgment from the receiver. In contrast, a *false-positive* detection error is usually caused by message loss. It detects a node as dead, while the node is still alive. We focus on addressing this type of detection error in the failover scheme.

Figure 4.5, extended from Figure 4.4, depicts the failure recovery scheme. As shown, the discussion of the failover falls into three parts, depending on a participating node is detected as failed (by gossip) *before*, *during*, or *after* the execution of rebuilding replicas of the partition P_i. In the following, T_{pause} is defined as a time period that is longer than twice the end-to-end gossip broadcast delay. This time period is effective in allowing the failed node to resurrect, because most temporary node failures are due to *false-positive* detection errors caused by message loss.

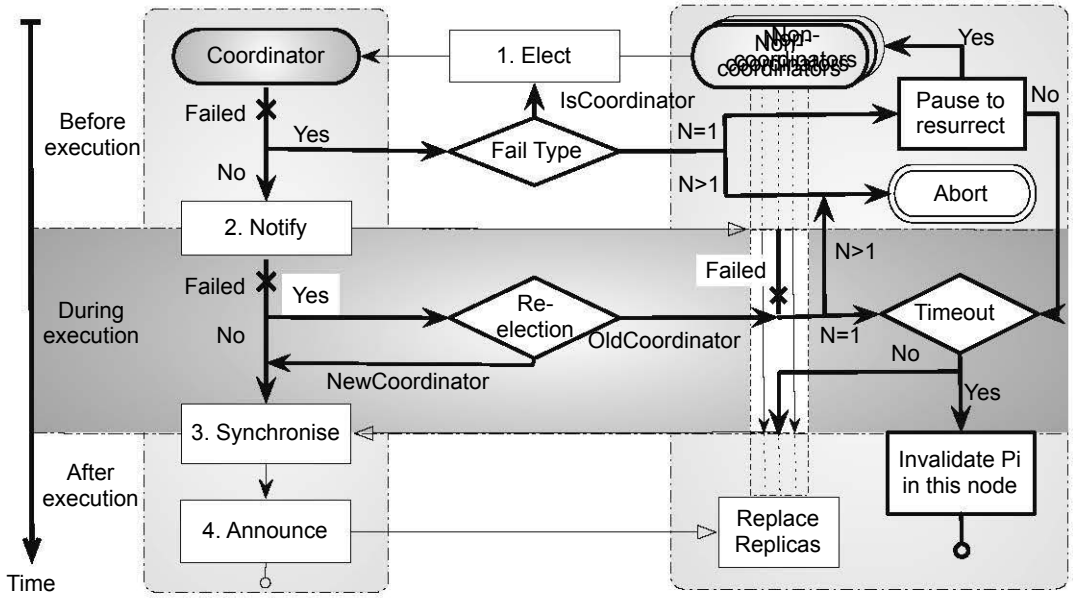


Figure 4.5: Failure recovery in the election-based coordination

Before the execution.

This phase starts when a partitioning operation is initiated, and ends before a notification for partitioning is sent out by the coordinator. In this period, if a participating node is detected as failed, the partitioning procedure is paused for T_{pause}. There are three scenarios following the pause. First, if more than one participating nodes are detected as failed, then the partitioning operation is aborted. Second, if only one participating node fails, and it is the coordinator, then a different coordinator should be elected, even when the previous coordinator resurrects. Third, if it is one non-coordinator that fails, then the partitioning operation should continue after T_{pause}.

In addition, the node that resurrects within T_{pause} can continue to participate in the operation, while the node that resurrects after T_{pause} is not allowed to participate. Yet, this node continues to serve queries for the targeted partitions, until the coordinator announces an update to the key range (at the end of Step 4). At this point, this node invalidates its own replicas, and it will no longer serve the targeted partitions.

During the execution.

This phase starts when the notification for partitioning is sent, and ends when all the *living* nodes have finished the execution of rebuilding replicas. During this period, if there is only one node (even it is the coordinator) that fails, the other participating nodes continue the operation of rebuilding the replicas regardlessly. However, if more than one participating nodes are detected as failed, and remain dead for a period of T_{pause}, then

the operation is aborted.

The challenge here is to determine the number of failed nodes if the coordinator itself fails. In the design, every participating node maintains the complete list of participating nodes, and is responsible for detecting node failures. Thus, any participating node that confirms the event of multiple node failures (i.e., after waiting for T_{pause}), can announce the abortion of partitioning via broadcast.

After the execution.

The execution of rebuilding replicas is considered complete when all the living nodes have finished the execution, with no more than one node failed during the execution. At the end of the execution, if there is one failed node, then the partitioning procedure is paused for another T_{pause} to await the node's resurrection. If the node resurrects within T_{pause}, the other nodes should await until the resurrected node finishes rebuilding its replicas. Otherwise, the failed node is excluded from the participation list.

Now that all the living nodes have been settled, the targeted partitions will be invalidated from any dead nodes. In addition, if the dead node is the coordinator, then a new coordinator is elected amongst the living nodes, following the same algorithm as in Step 1. Once the coordinator is in presence, it announces that the partitioning is successful, as in Step 4.

In any other scenarios that are not discussed above, the partitioning operation can be safely aborted. Such abortion does not incur any data loss, because the original partition replicas are still in use before the final announcement from the coordinator. The aborted partitioning operation will be reinitiated after a long pause. In addition, during the failover, some failed nodes have invalidated their own replicas. If they resurrect, these nodes have the priority to regain the ownership of the targeted partitions after the partitioning operation is completed or aborted.

Hence, this automated partitioning algorithm consolidates partitions that are suitable for efficient data migration. Based on such consolidated replicas, the next subsection discusses the replica placement strategy for node bootstrapping and decommissioning.

4.2.3 Replica Placement

This replica placement scheme is focused on the assignment of partition replicas to the nodes. It differs from, but does not contradict, the ElasticCopyset placement scheme

described in Chapter 6, which determines how to group the nodes for higher data durability at multiple node failures. Hence, both placement schemes can co-exist in a KVS. Listed is a number of rules that this placement scheme follows. The notational conventions used can be found in Table 4.1.

- **Rule 1: Complexity Reduction**. Partition reallocation and repartitioning are mutually exclusive. That is, partitions that are being split or merged will not be selected for reallocation, and partitions that are being reallocated will not be split or merged. This is to avoid the complex coordination between data placement and partitioning.

- **Rule 2: High Availability**. Each partition P_i has ν_i replicas allocated in ν_i different nodes. The replication number, denoted as K, is configurable by the system administrators. It is required that $\forall i \in [1, Q], \nu_i \geq K$, wherein Q is the number of partitions. If a partition has less than K replicas (e.g. due to node failure), a replica should be duplicated to a separate node.

- **Rule 3: Load Balancing**. The load on a node is reflected in its CPU usage, as the CPU is a load-dependent resource. The workload of node n_i is denoted as W_i. The nodes with higher workloads have higher priority to offer replicas. Hence, heavily loaded nodes have the priority to move out more replicas (thus shifting the workload) to the new node.

- **Rule 4: Data Balancing**. Since each partition replica is confined into bounded sizes by automated partitioning, balancing the number of partition replicas can result in balancing the volume of data stored in each node.

The rest of this subsection elaborates a two-phase data migration strategy to achieve quick node bootstrapping and load balancing, with Algorithm 1 and Algorithm 2 for populating a new node presented. Next, Algorithm 3 is presented as a strategy for redistributing replicas when a node is decommissioned. This discussion finishes with Algorithm 4 for dealing with node failures.

Node Pre-bootstrapping

In the *pre-bootstrapping* phase, the new node aims at maintaining high availability (referring to Rule 2) and alleviating nodes under heavy workloads (Rule 3). As shown in

Algorithm 1 Generate replica operations for the pre-bootstrapping phase

```
1:  ops ← newEmptyList()
2:  for i = 1 → Q do
3:      if IsPartitioning(P_i) = true then                ▷ Referring to Rule 1
4:          continue
5:      else if ν_i < K then                              ▷ Referring to Rule 2
6:          op ← replicate(P_i)
7:          ops.add(op)
8:      end if
9:  end for
10: {W_i}_{i=1}^N ← getWorkloadByNodes()
11: for i = 1 → N do
12:     if IsHeavyLoad(W_i) = true then                   ▷ Referring to Rule 3
13:         {P_j}_{j=1}^k ← getPartitionsFrom(n_i)
14:         total ← k * proportion
15:         count ← 0
16:         for j = 1 → k do
17:             if count ≥ total then
18:                 break
19:             else if containsPartition(ops, P_j) = false then
20:                 op ← migrate(P_j, n_i)
21:                 ops.add(op)
22:                 count ← count + 1
23:             end if
24:         end for
25:     end if
26: end for
27: return ops
```

Algorithm 1, the new node will skip all the partitions that are being split or merged (Rule 1). Next, for those partitions that do not meet the replication number K, it will make an extra replica (i.e., $replicate(P_i)$ in Line 6). After that, the new node pulls in a small proportion of the replicas from each heavily loaded node, defined by $IsHeavyLoad(W_i)$ in Line 12. Once the new node receives these replicas, it completes bootstrapping and starts serving queries immediately as a member of the KVS.

The CPU usage is collected to estimate the workload each node undertakes. W_i represents the moving average of the CPU usage in n_i. The CPU usage (i.e. W_i) is piggybacked on the heartbeat gossip message, sent by each living node periodically and cached by every other node. Therefore, a new node can download complete workload information from any existing node. This operation is referred to $getWorkloadByNodes()$ in Algorithm 1. A node is marked (by the new node) as *heavily loaded*, if its CPU usage exceeds a certain threshold, which is left for the KVS administrators to decide. One example of the threshold

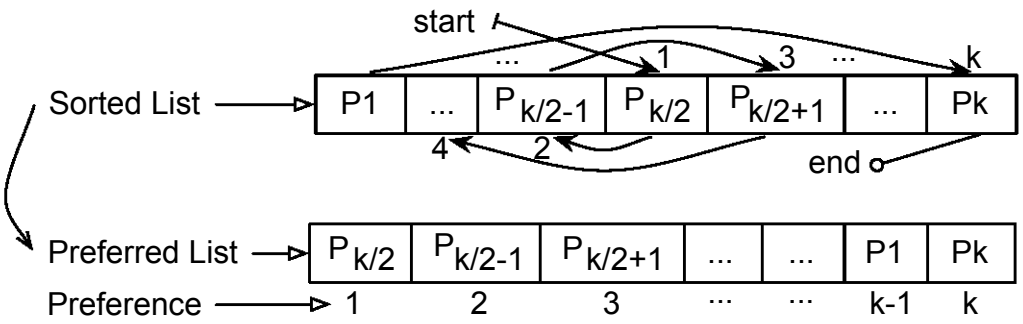


Figure 4.6: Prepare a preference list of partitions to move out

is that the CPU usage is over 50% and reasonably (e.g. 20%) greater than the average of all the nodes.

There are also considerations on how to select partition replicas, when an existing node is requested to offer data. Each node maintains an exponential moving average (EMA) of the *local* hit count for each replica, which is updated periodically as in Equation 4.3. For partition P_i at time t, the moving average of the local hit count is denoted as $H_{i,t}$, while the actual local hit count between time $t-1$ and t is denoted as $h_{i,t}$. The coefficient α represents the degree of weighting decrease, and is tuneable by system administrators. A higher value of α means a greater weight that $h_{i,t}$ contributes.

$$H_{i,t} = \alpha h_{i,t} + (1-\alpha) H_{i,t-1} \tag{4.3}$$

Figure 4.6 describes how to prioritise the orders for the partition replicas to move out. Firstly, the node sorts its own replicas by the EMA of hit count. Then, the pointer traverses the sorted list starting from the middle, and hops towards the left and right ends alternately. Thus, the hottest and coldest partitions are placed at the end of the preference list. This strategy is designed to avoid the greedy heuristic, which has two limitations. First, moving the hottest replicas will hurt the query performance during the migration, and may overwhelm the destination node. Second, moving the coldest replicas does not help shift the workload over the nodes, but only feeds a new node with large amounts of unpopular data.

When the new node receives such a preference list of partition replicas (i.e. via $getPartitionsFrom(n_i)$ in Line 13), it chooses a small proportion of partitions from the top of the list. The value of *proportion* (Algorithm 1, Line 16) is given by $proportion = minimum(1/N, 10\%)$, where N is the number of nodes. Thus, even if all the N nodes are heavily loaded, the new node will take over only an average number of replicas during the

Algorithm 2 Generate replica operations for the post-bootstrapping phase

```
Require: ops                                   ▷ Requires operation records from Algorithm 1
 1: R̄ ← Σ_{i=1}^{Q} ν_i/(N+1)                  ▷ Average #replicas each node will have
 2: {W_i}_{i=1}^{N} ← getWorkloadByNodes()
 3: nodeList ← getSortedList({W_i}_{i=1}^{N}, descend)        ▷ Referring to Rule 3
 4: for i = 1 → N do
 5:     node ← nodeList.get(i)
 6:     {P_j}_{j=1}^{k} ← getPartitionsFrom(node)
 7:     total ← maximum(k − R̄, 0)                ▷ Referring to Rule 4
 8:     count ← 0
 9:     for j = 1 → k do
10:         if count ≥ total then
11:             break
12:         else if IsPartitioning(P_j) = true then        ▷ Referring to Rule 1
13:             continue
14:         else if containsPartition(ops, P_j) = false then
15:             op ← migrate(P_j, node)
16:             ops.add(op)
17:             count ← count + 1
18:             if ops.size() ≥ R̄ then             ▷ Referring to Rule 4
19:                 return ops
20:             end if
21:         end if
22:     end for
23: end for
```

pre-bootstrapping phase. The value 10% is used to handle a small cluster with less than 10 nodes, in which case each node offers 10% of its replicas. In addition, when the new node traverses the list, it chooses those partitions that are not already selected for migration (i.e. not found in the *ops* list). Note that $migrate(P_j, n_i)$ moves partition P_j from node n_i to the node that executes this operation, while $replicate(P_i)$ duplicates partition P_i from any available replica.

Node Post-bootstrapping

In the *post-bootstrapping* phase, the new node has already been bootstrapped, and aims to achieve load-balancing in terms of both workload and data volume. The new node continues to pull in more replicas from other existing nodes, using the operation list generated by Algorithm 2.

To achieve balanced data volume, let $\overline{R}$ be the average number of replicas within each node. The new node recalculates $\overline{R}$ as in Line 1 of Algorithm 2. The new node will continue to pull in data as long as it has less than $\overline{R}$ replicas (Line 18), and an existing

node keeps offering (i.e. move out) replicas while it has more than $\overline{R}$ replicas (Line 7). These operations follow Rule 4, which balances the number of replicas that each node serves. Since a data set is split into many partitions of a bounded size, assigning an equal number of replicas is very likely to yield a similar volume of data in each node.

Moreover, in order to balance the workload, the new node prioritises the existing nodes by their workloads. As shown in Line 7 of Algorithm 2, the nodes with higher workload have a higher priority to offer replicas. Following the pre-bootstrapping phase, each node offers a preference list of partition replicas, sorted using the algorithm depicted in Figure 4.6. Additionally, the process of post-bootstrapping is run in a background thread, with data transfer rate throttled, so that the side-effects on online query processing are minimised.

Overall, in this two-phase procedure, the new node receives the majority of its replicas in the *post-bootstrapping* phase. The reason is that, in the *pre-bootstrapping* phase, each heavily loaded node offers only a small proportion of replicas, and in most cases, not every node is heavily loaded. By comparison, in the *post-bootstrapping* phase, the new node can pull in data from any node (no matter loaded or not) until it has acquired $\overline{R}$ replicas. Therefore, the volume of data transferred in *pre-bootstrapping* is reduced by a large extent, and the first phase of bootstrapping is completed in a timely manner. Hence, the new node is able to start serving queries within a short time after it is initiated, and therefore achieves quick bootstrapping.

Node Decommissioning

There are circumstances when node decommissioning is necessary. First, provisioned computing resources may become redundant due to decrease in workload demand. For example, none of the living nodes is *heavily loaded* and there exist nodes that receive less queries than expected. Second, a living node is misbehaving, e.g. it is failing more often than it should or its performance is noticeably slow. Third, the system administrators may reduce the system scale due to other strategic and operational considerations, such as shutdown for scheduled maintenance of infrastructure. In any case, the decision to decommission a node is made by the KVS administrators. This discussion is focused on how to reallocate the replicas when node decommissioning is requested.

Algorithm 3 presents the replica operations for node decommissioning. The node to be

Algorithm 3 Generate replica operations for decommissioning a node

```
1:  ops ← newEmptyList()
2:  {H_{i,t}}_{i=1}^{k} ← getHitCounts()                ▷ Sort replicas in descendant by hit count
3:  replicaList ← getSortedList({H_{i,t}}_{i=1}^{k}, descend)
4:  {W_i}_{i=1}^{N-1} ← getWorkloadByNodes()            ▷ Sort nodes in ascendant by workload
5:  nodeList ← getSortedList({W_i}_{i=1}^{N-1}, ascend)
6:  R̄ ← Σ_{i=1}^{Q} ν_i/(N − 1)
7:  for i = 1 → N − 1 do
8:      proportion ← getProportion(i, N − 1)
9:      maxR ← R̄ ∗ (1 + proportion)                    ▷ Allow extra replicas for idle nodes
10:     if replicaList.size() ≤ 0 then
11:         return ops                                  ▷ Move out until no replica left
12:     end if
13:     node ← nodeList.get(i)
14:     count ← getReplicaCount(node)
15:     for j = 1 → replicaList.size() do
16:         replica ← replicaList.get(j)
17:         if count ≥ maxR then                        ▷ Accept replicas until maximum
18:             break
19:         else if containsReplica(node, replica) = false then
20:             op ← moveTo(replica, node)
21:             ops.add(op)
22:             count ← count + 1
23:             replicaList.remove(replica)
24:         end if
25:     end for
26: end for
```

decommissioned moves out its replicas *one by one*, sorted in descending order of average hit count $H_{i,t}$ (Equation 4.3). The destination nodes that accept the replicas are prioritised based on their workloads in ascending order. And the nodes with a lower workload are allowed to accept an extra proportion of replicas than the average number of replicas that each node should have.

It is worth mentioning that Algorithm 3 assigns different proportions of replicas to nodes with different priorities. As shown in Line 8, the value of *proportion* is determined by the position of the node in the list. In the design, the return of $getProportion(i, N-1)$ is given by $(1 - \frac{i}{N-1}) * maximum(\frac{1}{N-1}, 10\%)$, which equals $maximum(\frac{1}{N-1}, 10\%)$ when $i = 0$, and decreases to 0 when $i = N - 1$. Similar to the proportion value in node bootstrapping, the function $maximum(\frac{1}{N-1}, 10\%)$ is used to deal with a small cluster of less than 10 nodes. In this way, the nodes with lower workloads receive approximately 10% more replicas than they should have, while the nodes with higher workloads may not receive any replica, because the decommissioning node may have moved out all its replicas

(Line 11).

These prioritisation prefers to balance the query workload rather than the data volume, because storage is much cheaper than the computation on the Cloud. Overall, the replicas from the decommissioning node are scattered over many living nodes, and the nodes with a lower workload receive more replicas that are popular in terms of hit count. In the end, the node can safely leave the KVS when there are no more replicas under its ownership.

Recovery from Node Failure

Node failure is common in distributed systems, usually caused by hardware malfunction, network failure or even human misbehaviour. Although in certain circumstances multiple nodes may fail simultaneously, this discussion is focused on dealing with single node failures that occur unexpectedly.

To continually serve queries during node failure, hinted handoff (DeCandia et al., 2007) is used. To illustrate, when a node n_i is unreachable for a write, a copy of the write operation is sent to another node n_j that does not currently serve the partition for the write. This is to maintain the desired availability guarantees (i.e., at least K replicas). This destination node n_j is termed as the **surrogate node**, and will write a hint in its metadata, suggesting that the write was intended for n_i. Once n_j detects that n_i has resurrected, it will attempt to deliver the copies it saved back to n_i. Finally, the surrogate node n_j can safely delete the copies once the transfer succeeds in n_i.

However, hinted handoff is intended for transient node failures. If the failure becomes permanent, additional replicas have to be created for all the partitions that were hosted by the dead node, so that the availability requirements (i.e. Rule 2) are met. There are two approaches to provide recovery from permanent failure. One strategy is to bootstrap a new empty node. In the pre-bootstrapping phase, this new node will make an extra replica for those partitions that do not meet availability requirements (shown in Algorithm 1, Line 6). Alternatively, the existing nodes have to create the replicas among themselves, which is termed here as **spontaneous recovery**.

The procedure of *spontaneous recovery* follows the same coordination described in Subsection 4.2.2. It is initiated by a surrogate node of hinted handoff (i.e. n_j in the previous discussion). There can be multiple surrogate nodes in presence, since the dead node was serving multiple partitions. When one of the surrogate nodes decides that a

Algorithm 4 Generate replica operations for spontaneous recovery

```
1: ops ← newEmptyList()
2: partitionList ← newEmptyList()
3: for i = 1 → Q do
4:     if IsPartitioning(P_i) = true then                    ▷ Referring to Rule 1
5:         continue
6:     else if ν_i < K then                                  ▷ Referring to Rule 2
7:         partitionList.add(P_i)
8:     end if
9: end for
10: {W_i}_{i=1}^N ← getWorkloadByNodes()                     ▷ Sort nodes in ascendant by workload
11: nodeList ← getSortedList({W_i}_{i=1}^N, ascend)
12: R̄ ← (partitionList.size() + Σ_{i=1}^Q ν_i)/N
13: for i = 1 → N do
14:     proportion ← getProportion(i, N)
15:     maxR ← R̄ * (1 + proportion)                          ▷ Allow extra replicas for idle nodes
16:     if partitionList.size() ≤ 0 then
17:         return ops
18:     end if
19:     node ← nodeList.get(i)
20:     count ← getReplicaCount(node)
21:     for j = 1 → partitionList.size() do
22:         partition ← partitionList.size(j)
23:         if count ≥ maxR then                             ▷ Accept replicas until maximum
24:             break
25:         else if containsPartition(node, partition) = false then
26:             op ← replicateTo(partition, node)
27:             ops.add(op)
28:             count ← count + 1
29:             partitionList.remove(partition)
30:         end if
31:     end for
32: end for
```

node is permanently dead (e.g., after waiting for a long time), it initiates a public vote that requires all the living nodes to elect a coordinator.

The coordinator is responsible for generating a list of replication tasks based on Algorithm 4. This algorithm shares a very similar logic to the distribution of replicas for node decommissioning (Algorithm 3), except that the coordinator does not prioritise the partitions for replication. Another difference is that, in node decommissioning, the replicas are moved out to the other living nodes (Algorithm 3, Line 20); while in *spontaneous recovery*, the coordinator requests a less loaded node to replicate the partition from other living nodes (Algorithm 4, Line 26).

As can be seen, each replication task consists of a partition to be replicated and a

destination node to host the replica. The coordinator assigns the tasks to each related node, and supervises the execution of these tasks until they are completed. The node that executes a replication task is free to choose the source node from which the partition is replicated. It finishes by sending an *ack* back to the coordinator. Once the coordinator has collected the acknowledgement of completion for all the tasks, the procedure of *spontaneous recovery* is successfully completed.

As in any distributed process, there is a probability of failure during *spontaneous recovery*. The coordinator may fail during the process, and is replaced by re-election. A new coordinator can decide whether to regenerate a list of replication tasks or continue to supervise the existing tasks. Besides the coordinator, a node that execute the replication tasks may also fail. In this case, the coordinator will submit another list of replication tasks after all the living nodes have acknowledged their completion.

4.2.4 Data Migration

In the previous subsection, the replica placement scheme generates a list of replica operations, each consisting of a partition P_i to operate, a source node n_s to offer data, a destination node n_d to receive data, and the operator, which can be either *migrate* or *replicate*. The challenge of executing these operations is to maintain data consistency guarantees during data migration. This subsection first introduces the consistency model, followed by a token ownership policy that is tailored to the consistency model. Last but not least, it presents a proof that demonstrates the data consistency is maintained.

Consistency Model

The consistency model here follows that of Cassandra (Apache, 2009) wherein KVS users define a consistency level L for each individual query operation. Different consistency levels require different numbers of nodes to acknowledge before a query operation is considered valid. A higher consistency level needs more nodes to respond, thus increasing the request latency, while a lower consistency level requires fewer nodes to reply, but is prone to reading stale value and losing a recent write. Therefore, the users can trade consistency guarantees against request latencies.

It is worth mentioning that, the consistency level L is independent from the replication number K (Subsection 4.2.3, Rule 2). The replication number requires each data object

Types of node set	**Number of nodes**	**Responsibility**	**Time to respond**	**Query type**
L set	$\geq$ the maximum consistency level	Serve online query for P_i	Immediately	Both read and write
K set	$\geq$ the replication number	Maintain replicas for P_i for availability	Eventually	Delayed write, no read

Table 4.2: The sets of nodes entitled to serve query operations for a partition P_i

to be *eventually copied* into K replicas for the concerns of availability and durability. In contrast, the consistency level requires L nodes to respond *before* a query operation is committed. Hence, $L \leq K$. Regarding to a write operation, the first L nodes that have acknowledged the write, will propagate the value to all the K nodes eventually, i.e., to provide *eventual consistency*.

From the perspective of query processing, the group of nodes that are entitled to achieve the consistency level L for certain partition, is termed as the L set, while the group of nodes entitled to maintain the K replicas, is termed as the K set. There are implications regarding these two sets, summarised in Table 4.2.

Being in an L set means that, a node is capable of serving both read and write operations for the related partition *at any time*. Hence, it requires that the latest writes be propagated to this node, which means that, a node in the L set also belongs to the K set of the related partition. Conversely, being in a K set does not make the node responsive to query operations. It only means the writes of the related partition should be propagated to this node eventually, for the sake of availability and durability. In other words, the K set of nodes should store data of the related partition, but do not necessarily serve online query. In almost all the KVSs, the L set and the K set are the same set of nodes that serve the partition, except in the our proposed scheme of data migration, where the L set is a subset of the K set, i.e. $L\ set \subseteq K\ set$.

Token Management for Data Migration

A token ownership policy is proposed to ensure data consistency for partition replicas that are being migrated or replicated. As discussed in Subsection 4.2.1, each partition P_i is associated with one token T_i. For query execution purposes, only the nodes that are entitled to host P_i can own the token T_i. Moreover, each token is a boolean value, wherein a *positive* token indicates that the node is in both the L and K sets, while a *negative*

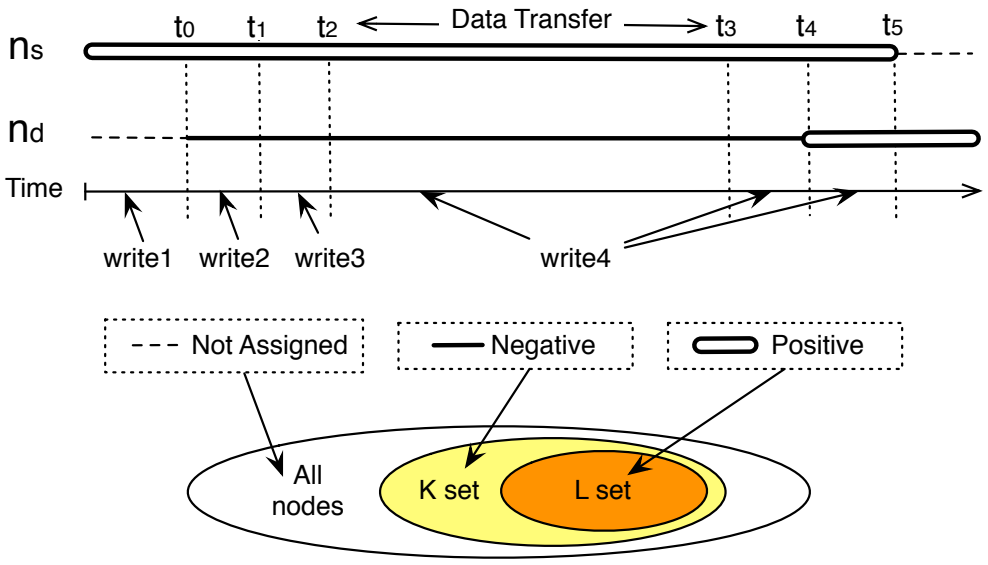


Figure 4.7: Manage a token during replica migration for data consistency

token allows the node to be in the K set only.

Figure 4.7 depicts when and how to assign and switch the value of T_i, when a replica of the partition P_i is being migrated from n_s to n_d. As shown, a request for migrating P_i is submitted at time t_0. The source node n_s makes a snapshot of P_i at t_2, immediately transfers the snapshot (i.e. replica) to the destination node n_d, and finishes the transfer at t_3. Each time interval between t_i and t_{i+1} is longer than the end-to-end gossip broadcast delay, so that a message for an update of token value is well propagated.

In terms of token management, the source node n_s owns the *positive* T_i before the data transfer is completed at t_3, because it owns P_i and is entitled to serve online queries for this partition. In contrast, the destination node n_d does not own P_i (or T_i) before t_0. At time t_0, the migration task of P_i is initiated, and n_d is assigned with the *negative* T_i. Having a *negative* T_i means that, the ongoing writes destined to P_i will be propagated to n_d in the background (by other nodes with the *positive* T_i). It also means that n_d does not serve any query, because this node does not have the complete replica for P_i. The gossip message about n_d having the *negative* T_i is spread to the whole network by the time t_1. From t_1 onwards, n_d is known by the other nodes to be entitled to store the writes destined to P_i.

The source node n_s waits until t_2 to start data transfer, so as to leave a sufficient time window for n_d to receive all recent writes dating from t_1. The transfer finishes at t_3. On receiving the complete replica, n_d takes a short interval to merge the writes that are

propagated to it between t_1 and t_3, into the replica. The destination node n_d finishes the merging of writes at time t_4, so it possesses an updated replica of P_i just like the other nodes that own the *positive* T_i. Hence, n_d switches T_i from negative to positive, and starts to serve online queries for P_i. The message for this token switching is gossiped throughout the KVS network. At time t_5, the source node n_s hears that n_d has taken over P_i, so it releases its own token T_i and discards its local replica safely.

In comparison, the operation of replicating replicas (e.g. from n_s to n_d) is very similar to migrating replicas. The only difference is that, at the end of data transfer, the source node n_s does not release T_i (or discard the replica), because the intention of replication is to create an extra replica for P_i.

Consistency during Data Migration

A proof by contradiction is presented to demonstrate that the replica in the destination n_d is consistent with the other existing replicas of P_i. To begin with, we presume that the data is inconsistent, which means, there exists a recent write that is not saved to n_d. There are four possible scenarios of a committed write, depicted as *write1*, *write2*, *write3*, and *write4* in Figure 4.7.

- The write is committed before t_0, i.e. *write1*. Then n_s has enough time (i.e. $t_2 - t_0$) to save the write into its replica, which is transferred to n_d at t_2. Thus, n_d sees *write1* at t_3, that is, before n_d starts to serve queries at t_4.

- The write is committed between t_0 and t_1, i.e. *write2*. Then this write may or may not be propagated to n_d, depending on when the negative T_i is seen. But n_s has enough time (i.e. $t_2 - t_1$) to save the write into its replica before transfer. Thus, n_d still sees *write2* at t_3.

- The write is committed between t_1 and t_2, i.e. *write3*. Then, n_s may not have enough time to save this write into its replica before transfer. However, *write3* is sure to be propagated to n_d, because the other living nodes already know that n_d is entitled to store a replica for P_i. Hence, n_d sees the latest version of *write3* when the write is propagated to it. At t_3, n_d may also see another version of *write3* in the replica from n_s. The conflicts between different versions are reconciled based on the timestamp stored in the data object, and "the last write wins" (Thomas, 1979).

- The write is committed after t_2, i.e. *write4*. Then, n_s does not save this write into the replica (i.e. a snapshot of P_i) that is transferred, but *write4* is sure to be propagated to n_d. After n_d receives the replica at t_3, it sees its own version of *write4*, and it reconciles any version conflicts seen based on the timestamp.

Hence, all the possible scenarios of writes are guaranteed to be saved to the destination node n_d. It contradicts the previous presumption, and thereby proves that the replica migrated to n_d is consistent with the other replicas of P_i in the KVS.

Overall, the process of online query is not interrupted by the migration of replicas. The source node n_s is responsible for serving P_i, along with the other nodes having the *positive* T_i, during the whole process of data transfer. This avoids interruptions to query processing, with two time windows to hand over the ownership of the partition. One time window is before the transfer, i.e. from t_0 to t_2, during which the destination node n_d becomes eligible to accept writes destined to P_i. The other time window is after the transfer, i.e., from t_3 to t_5, when n_d possesses the complete replica and is made known to all nodes that it starts to serve online queries for P_i. These two time windows guarantee a consistent view of the partition-node mappings for all the KVS nodes.

4.3 Chapter Summary

Efficient elasticity is an important feature for distributed KVSs running virtual machines (VMs) leased from IaaS Cloud. In order to improve the efficiency of elasticity for shared-nothing KVSs, this chapter has presented a set of decentralised data management schemes. It started with an automated partitioning algorithm, which splits and merges partitions based on the actual data volume in each partition. With automated partitioning, each partition replica is confined into a bounded size, and is consolidated into a transferable unit, so that the overheads of data migration at node changes are reduced, and load-balancing also becomes simpler.

Moreover, this chapter has presented an election-based, fault-tolerant coordination to facilitate the execution of automated partitioning in a decentralised manner. It is followed by the description of four replica placement algorithms that determine the assignment of replicas to the nodes for the scenarios of two-phase bootstrapping, decommissioning, and unexpected failure, respectively. This placement scheme achieves fast bootstrapping,

and maintains well balanced workload and distribution of data. Finally, it has depicted the execution of replica placement decisions, i.e., data migration, which leverages a token ownership policy to maintain online query processing during data migration, and to provide eventual consistency.

This set of decentralised data management schemes forms a middleware layer between the key space of a data set and the storage of the KVS nodes. The major innovations in this middleware are to automatically partition the data into transferable units for decentralised KVSs, and to achieve efficient node bootstrapping and decommissioning with the help of the replica placement and migration schemes that build on the transferable replicas.

The next chapter will discuss the implementation of this middleware layer, to build an elastic KVS called ElasCass on top of Apache Cassandra. The evaluations of ElasCass on a public IaaS Cloud will also be presented.

Chapter 5

Implementation and Evaluation of ElasCass

Idle hands are the devil's workshop.

– St. Jerome

The previous chapter proposed the design of a set of data distribution schemes for the efficient elasticity of decentralised shared-nothing KVSs. To realise the proposed schemes, this chapter presents the implementation and evaluation of **ElasCass** (Elastic Cassandra), which is built on top of Apache Cassandra. It starts with an introduction to the challenges of implementing the proposed data distribution schemes, and then compares a number of open source KVSs and describes the background of Apache Cassandra. Next, it presents the three core functionalities implemented in ElasCass. Based on the implementation, this chapter presents the experimental evaluations of ElasCass against Apache Cassandra, and then concludes.

5.1 Introduction

Distributed KVSs have become a reference architecture for managing large volumes of data on servers leased from the IaaS Cloud. The key feature of the Cloud is resource elasticity, which requires a KVS to bootstrap or decommission a node quickly and frictionlessly. The previous chapter has targeted the decentralised shared-nothing KVSs, and proposed the design of a set of data management schemes in this regard. This chapter focuses on the implementation and evaluation of the proposed design.

There have been a number of open source KVSs that follow the decentralised shared-nothing architecture introduced in Amazon's Dynamo (DeCandia et al., 2007), such as Cassandra (Apache, 2009), Riak (Basho Tech., 2012), and Project Voldemort (2009). Therefore, leveraging an established KVS is more feasible for implementing the proposed data management schemes than developing a new KVS from scratch. We have identified Cassandra (Apache, 2009) as the basis of our implementation due to several reasons. First and foremost, it follows the decentralised shared-nothing architecture, which matches our system prototype described in Figure 4.1 (page 70). Second, it derives many design choices from Google's Bigtable (Chang et al., 2006) and Amazon's Dynamo (DeCandia et al., 2007), both of which are influential KVSs that have demonstrated their success in serving large volumes of data for interactive web applications and services. Third, it is an open source software under the development of Apache Software Foundation with an active community around it.

However, there are challenges to realise our proposed design on top of Apache Cassandra. First, our design requires more sophisticated bindings between the partitions and nodes, while Cassandra follows the basic consistent hashing (Karger et al., 1997) to maintain a simple one-to-one mapping. Second, our design demands the consolidation of each partition replica into one standalone transferable unit, whilst in Cassandra there does not exist a mapping between data files and partitions. Third, our design proposes a set of replica placement algorithms based on workload statistics, while Cassandra uses static assignment. Last, our design relies on election-based coordination to perform a list of distributed tasks, which is not implemented in Cassandra.

This chapter presents the implementation of *ElasCass*, which addresses the challenges described. The implementation consists of a token management component that deals with dynamic partition-node mappings during automated partitioning and replica reallocation, a data storage component that efficiently confines the files into standalone partition replicas with a background process, and a replica reallocation component that gathers workload statistics and coordinates nodes for ensuring consistency during data movement. In addition, the election-based coordination is integrated with the three components described.

This chapter also presents a set of experimental evaluations, carried out using a public IaaS Cloud, that demonstrates that the proposed schemes of data partitioning and real-

location: i) distribute data and workload more evenly among the nodes, ii) reduce the time to bootstrap nodes and improve data movement speed, and thereby, iii) improve the scalability and throughput of the KVS.

5.2 Background

5.2.1 Existing Open Source KVSs

The aim of this implementation is to build a KVS on top of an existing open source project, so as to realise the data distribution schemes proposed in Chapter 4. Compared to developing a brand new KVS from scratch, leveraging an established KVS helps us concentrate on the data management techniques, and also facilitates benchmarking against other current-state KVSs.

KVSs that have gained wide attention include Memcached (Danga Interactive, 2004), Redis (2009), Cassandra (Apache, 2009), Project Voldemort (2009), and HBase (Apache, 2010). Among the variety of choices, we intend to focus on those KVSs following the decentralised, shared-nothing architecture that is depicted in Figure 4.1 (page 70). As discussed, this architecture was adopted by Amazon's Dynamo (DeCandia et al., 2007), which is proprietary. However, it inspired a number of open source KVSs, including Riak (Basho Tech., 2012), Project Voldemort (2009), and Cassandra (Apache, 2009). A comparison among these three KVSs is presented as follows.

Riak (Basho Tech., 2012) follows many design choices in Dynamo. It inherits the virtual-node approach (Section 3.2.1, page 58), to split the key space into partitions with an equal-length key range. It retains the simple data model of key-value pairs, wherein the values are stored on disk as binaries. Riak also has a pluggable backend storage, with the default being Bitcask (while Dynamo uses Berkeley DB and MySQL as the backend storage). Bitcask is a log-structured hash table for write-once, append-only queries. There are two drawbacks in Riak. First, the data model of simple key-value mapping is too limiting for supporting structured and semi-structured data. Second, the log-structured Bitcask is not efficient in reads, which is compensated by retaining the whole key space in memory, and therefore, is not memory conservative.

Project Voldemort (2009) is very similar to Riak. It also uses the virtual-node approach for data partitioning and replication, and inherits the simple key-value data model. As

in Dynamo, the storage engines are pluggable, with the default being Berkeley DB and MySQL. One notable feature of Voldemort is the in-memory caching coupled with storage system, so that a separate caching tier is no longer required. The other feature is that several components are made pluggable, such as serialisation, data placement, and storage engine. However, the query API that supports only simple CRUD operations of key-value pairs, is not pluggable. This has limited its range of applications.

In contrast to Riak and Voldemort, Cassandra (Apache, 2009) adopts the model of column family from Google's Bigtable (Chang et al., 2006). Column family is richer than the simple key-value model, and is able to support structured and semi-structured data. Yet it is still simple enough to be efficiently stored in flat-file representation. Cassandra experienced a major change in the strategy of data partitioning between versions. Prior to Version 1.2 (which is released on Jan 18, 2013), it followed the traditional consistent hashing to partition the key space based on the number of data nodes, that is, the split-move approach described in Section 3.2.1 (page 57). This approach hurts load balancing. From Version 1.2 onwards, it has adopted the virtual-node approach similar to Dynamo, Riak and Voldemort. However, our implementation started in 2011. At that time Version 1.0 was the latest release, which adopted the split-move approach.

Cassandra is a mixture of Google's Bigtable and Amazon's Dynamo. Its column family model supports a wide range of applications, while its decentralised, shared-nothing architecture offers inherent scalability and the potential to be elastic. According to a recent survey of DB-Engines (2014c), Cassandra is indeed ranked top of the list of all kinds of KVSs. Hence, out of the three KVSs discussed, we have chosen Cassandra as the basis of our implementation.

5.2.2 Overview of Apache Cassandra

Apache Cassandra is a decentralised, shared-nothing KVS that handles large amounts of data across many commodity machines. Cassandra combines Dynamo's DHT-like architecture (DeCandia et al., 2007), with Bigtable's data storage schemes (Chang et al., 2006). Since Cassandra is an ever-developing project, an implementation choice in one release may be deprecated in a later release. To avoid inter-version confusion, this subsection mainly focuses on Cassandra Version 1.0 that served as the stable release between October 2011 and July 2012, during which ElasCass was implemented.

Architecture

Apache Cassandra follows Amazon Dynamo's DHT-like architecture (DeCandia et al., 2007). A Cassandra *cluster* contains multiple nodes, which form a P2P, symmetric network without the presence of any centralised component. Each node is running on one separate commodity machine, or a VM with a persistent storage device attached individually. The nodes do not share memory or storage space between each other, and hence, are deployed in a shared-nothing architecture.

Cassandra (version 1.0) leverages the traditional consistent hashing to distribute the data across multiple storage nodes. In consistent hashing (Karger et al., 1997), the largest key wraps around to the smallest key to form a fixed circular space or "ring". In a Cassandra cluster, each node is assigned a random value (i.e. *token*) within this space which represents its "position" on the ring. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The node associated with the token is also termed as the *coordinator* of the related partition. The responsibility of the coordinator is to maintain a master replica and to decide the surrogate node when a node storing a slave replica temporarily fails.

Data Distribution

Figure 5.1 depicts how Cassandra distributes its data when a node is added or removed from the cluster. **Scenario 1** shows that, each node serves as the *coordinator* (i.e. storing the master replica) for only one partition, the slave replicas of which are stored on the successor nodes (on the ring) following the coordinator, so as to improve data availability. For example, Node a is the coordinator of Partition A, which has two slave replicas on Node b and Node c.

Scenario 2 illustrates how to handle node arrival in Cassandra. A new node is assigned to a new position on the ring, for example, $c1$, which is between b and c, and splits Partition C into $C1$ and $C2$. The new node, associated with $c1$, becomes the coordinator for Partition $C1$. Since the new node is the successor of Node a and b, it also serves a slave replica for Partition A and B, the data of which is replicated from the corresponding coordinators. In contrast, Node c retains the coordinator ownership for Partition $C2$, the other half of Partition C. Moreover, since the new node has occupied Position $c1$ on the ring, its successors, namely Node c, d and e, are no longer responsible for Partition A, B

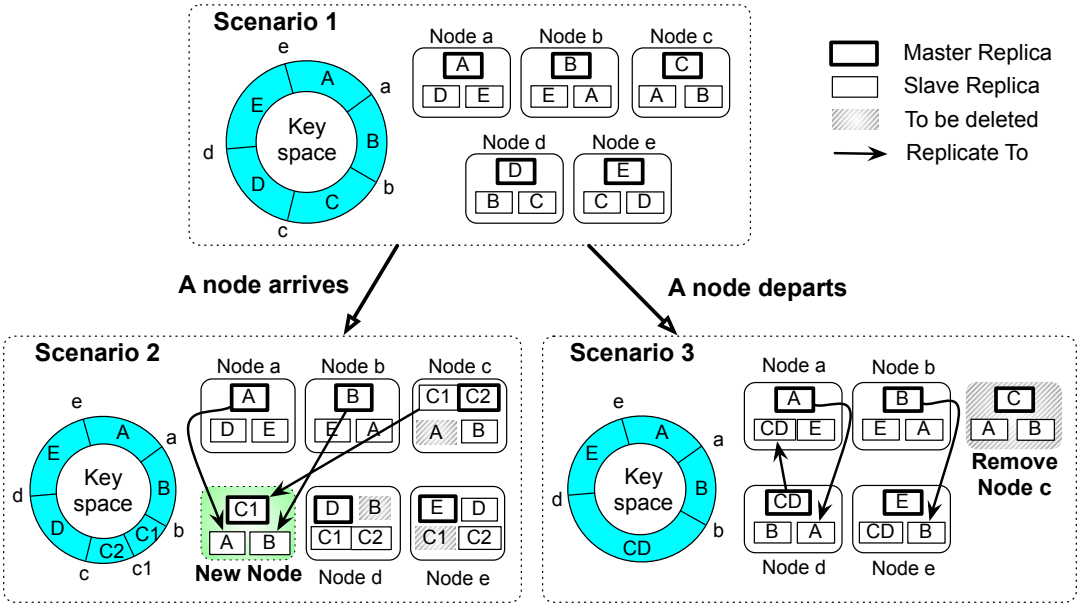


Figure 5.1: Data distribution at node changes in Apache Cassandra

and $C1$, respectively.

Conversely, **Scenario 3** show how to deal with node departure from Scenario 1. An existing node, e.g. Node c, fails or is decommissioned. Since it is associated with Token c on the ring, this token is no longer valid after the departure of Node c. As a result, Partition C and D are merged into one partition, i.e. Partition CD. The direct successor of Node c, i.e. Node d, becomes the coordinator for this merged partition. Moreover, since the replicas of Partition A, B and C are removed along with Node c, the affected partitions are replicated (from the corresponding coordinators) to Node d, e and a, respectively.

As can be seen, each node change affects its direct successor nodes, the number of which is equal to the number of replicas each partition has. Although this partitioning approach leverages consistent hashing (Karger et al., 1997) to affect a minimum number of nodes, it is not efficient in data migration, as discussed in Section 3.2.1 (page 57).

Data Storage and Access

Apache Cassandra has adopted many design choices from Google Bigtable (Chang et al., 2006) in terms of data storage and retrieval, including the data model of column family, described in Subsection 3.1.1 (page 47), and the processing of read/write operations (Figure 5.2).

A column family resembles a row-based table in databases. Each data object is a key-value mapping, wherein the *key* is the output of certain hash function (either

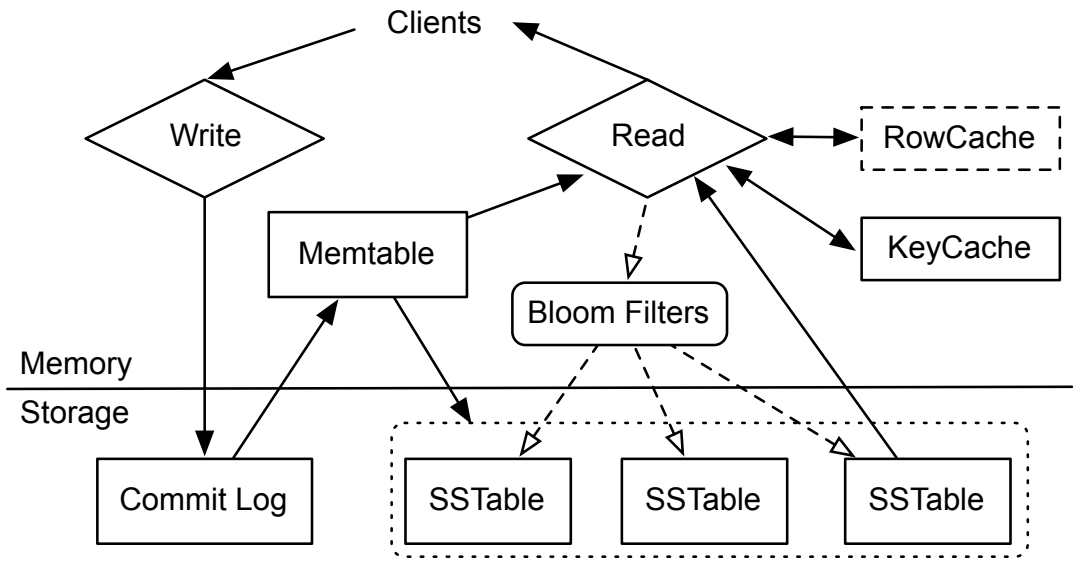


Figure 5.2: The read/write operations in Apache Cassandra

MD5/MurmurHash or lexical), and is serialised into an array of raw bytes. The *value* of a data object is a collection of standard columns, each consisting of a column name, a column value and a timestamp. There is also the super column, which is composed of a column name and a collection of standard columns. In Cassandra, multiple column families share the same partitioning of the *keyspace*, which resembles a database in database systems.

Figure 5.2 depicts how to handle the read/write operations in a column family. Each write (including insert, update, and delete) is considered *committed*, as soon as the operation is appended to a write-ahead commit log and the value is buffered in *memtable*, which is an in-memory bucket. The memtable is associated with one column family. When the memtable size reaches a threshold, the memtable is frozen, and is replaced by a new empty memtable in an atomic operation. The frozen memtable is converted to an SSTable, which is written onto the disk. The *SSTable* is an immutable, sorted string data file. Multiple SSTables (of the same column family) are regularly compacted in the background into one larger SSTable, so as to reduce metadata and file handlers, and more importantly, to remove obsolete data objects, including the ones marked as deleted and those overwritten by a newer version.

In contrast, each read operation attempts to retrieve the value from multiple data sources. First, it looks up the in-memory cache, including a *KeyCache* that contains the location of the related data object on disk, and an optional *RowCache* that caches query

results, which is memory-consuming. Second, it checks with *memtable* to see if the related data object is recently updated. Last but most important, it retrieves the data object from a list of on-disk SSTables. Note that the SSTables are immutable, and thus can store obsolete data objects. It requires that the node visit every SSTable of the column family so as to retrieve all the possible versions of certain data object. In order to reduce the number of I/O accesses, each SSTable has a Bloom filter (Bloom, 1970), which is loaded into the memory and is used to test the presence of certain data object given the key.

Overall, Cassandra is optimised for write operations with the use of append-only commit log and memtable, while it is less efficient in read operations since it allows obsolete data objects to scatter across multiple SSTables, making the retrieval of a data object less straightforward.

Query Processing

Since Cassandra leverages consistent hashing for data partitioning, each node is associated with a token which represents a position on the ring of the key space. To locate a data object given a key, simply walk the ring clockwise to find the first token larger than the key. As discussed, this token and its predecessor define the key range of the partition to which the given key belongs. Since each partition has multiple replicas for high availability, for the sake of query processing, each token is therefore mapped to the nodes that store a replica for the related partition. In addition, it is also worth mentioning that every node caches the mapping between the partitions and the nodes, so that a key lookup can be performed by any living node. The update of the mapping is propagated based on gossiping protocols (Lin & Marzullo, 1999).

Cassandra allows tunable consistency for each query. Different consistency levels demand for different numbers of nodes to acknowledge the query before a query is considered successful. Since each token is mapped to all the nodes serving the associated partition, the node connected to the client randomly selects K healthy nodes from the mapping, wherein K is the required consistency level of the query.

Cassandra also uses hinted handoff (DeCandia et al., 2007) to provide extreme write availability when a node temporarily fails. When the number of living (or healthy) nodes is less than the required consistency level (due to node failure), a hint is written to the coordinator (which serves the master replica), indicating that the write needs to be replayed

to the unavailable node.

Current State

To this end, the latest release of Apache Cassandra is Version 2.0. Two major changes have been made in the design since Version 1.0.

First and foremost, Cassandra has also adopted the concept of "virtual node" in Version 1.2, which was released in January 2013. It allows the developers to define the number of tokens before the key space is initialised, so that the key space is statically partitioned into the desired number of *vnodes*. This approach has no difference from Riak and Project Voldemort as discussed. By comparison, ElasCass not only uses the "virtual node" approach that Cassandra Version 1.2 has adopted, but also automatically splits and merges the partitions for more balanced distribution of data.

Second, Cassandra has provided row-level (i.e. data-object-level) transaction support in Version 2.0, released in September 2013. It extends the Paxos consensus protocol to provide linearisable consistency, and exposes this functionality as a compare-and-set operation. However, this linearisable consistency is provided at the cost of four round trips between a leader node and the replicas. Hence, it is only applicable for a small minority of operations in the system.

5.3 ElasCass Implementation

Table 5.1 summarises the differences between Apache Cassandra and ElasCass in terms of data management strategies and components. As shown, the strategies regarding data partitioning and placement are diverse in the two KVSs, and therefore it requires a significant amount of reprogramming to implement the functionalities related to the components of key space, token metadata, and SSTable in ElasCass. Conversely, in order to clarify, there are several functionalities that are leveraged from Apache Cassandra, including: i) the propagation of gossip messages; ii) hinted handoff for handling temporary node failures; and iii) the storage format of SSTable, that is, each SSTable has a Bloom filter, an index, and the compression information.

Table 5.1 also unveils the challenges in implementing ElasCass atop Cassandra, from the internal data storage and token management, to the inter-node data reallocation and coordination. A discussion is presented as follows.

Properties	Apache Cassandra	ElasCass
Strategies		
Data Partitioning	Each partition is associated with one node, and is split/merged at each node arrival/departure.	The key space is initially segmented into Q partitions, and is further split/merged depending on the volume of data.
Data Placement	Based on consistent hashing, the coordinator serves the master replica, while the successor nodes serve the slave replicas.	Each replica is assigned independently based on a set of placement algorithms.
Data Movement	Scan to prepare a list of data objects from multiple SSTables at source node; Rebuild SSTables at the destination node.	Simply transfer SSTables that are associated with certain partition.
Coordination	A node is the coordinator for a partition if it owns the associated token.	A coordinator is nominated via an election when required.
Components		
Key Space	Multiple column families share a key space.	Each column family has a separate key space for data partitioning.
Token Metadata	An array of raw bytes representing a position on the key ring; The token-node mapping is one-to-one.	Raw bytes for positioning, plus a boolean value indicating the node set for query processing (Table 4.2); The token-node mapping is many-to-many.
SSTable	An SSTable stores any data objects in the same column family.	An SSTable only stores the data objects belonging to one particular partition.
Memtable	Each memtable is flushed to form one new SSTable.	Each memtable is flushed to form multiple new SSTables, each associated with a different partition.

Table 5.1: Comparison between Apache Cassandra and ElasCass

- **Token management**. Data partitioning in Cassandra is based on the number of nodes, while ElasCass partitions the key space based on the data volume in each partition. Hence, unlike Cassandra that uses a unified key space for all the column families, ElasCass requires column-family-specific key spaces, since the data distributions in different column families are usually diverse. More importantly, ElasCass maintains flexible, many-to-many partition-node mappings, which requires a dynamic token management scheme that is far more complicated than the one implemented in Cassandra.

- **Data storage**. Cassandra stores data objects into any SSTable (i.e. a data file) of the same column family, while ElasCass requires that each SSTable be tied up with one specific partition, so that data movement can be easily executed by moving the integrated SSTables across the nodes. It gives rise to two issues. First, there is a need to maintain a map between a set of SSTables and a partition. Maintaining such a mapping is non-trivial, especially when the partitions are split or merged dynamically at runtime. Second, for the sake of efficiency, the SSTables within each partition should be compacted periodically, which is challenging in two ways. First, compactions should not affect online query processing. Second, compactions should be efficiently rolled back for fail recovery.

- **Data reallocation**. In Cassandra, the locations of data objects are relatively static, and are determined by the node positions on the key ring. By comparison, ElasCass treats each partition replica as a transferable unit that can be dynamically reallocated to any node as required. This requires the implementation of a set of replica placement algorithms, and a data migration scheme that transfers the data in the form of SSTables, without affecting the online query performance or impinging on data consistency.

- **Coordination**. In Cassandra, each node is associated with one token that defines one partition. Thus, each node serves as the coordinator for the partition with which it is associated. In contrast, ElasCass requires the Chubby service (Burrows, 2006) for the election of a coordinator within a set of nodes (i.e. not necessarily the whole cluster of nodes). Since the mechanism of the coordination has been thoroughly discussed in Subsection 4.2.2 (page 75), the focus will be on message exchange, which is integrated with the implementation of the previous three components.

In the remaining of this section, three subsections will be presented to describe the implementation of three core functionalities in ElasCass, namely token management, data storage, and data reallocation.

5.3.1 Token Management

In order to realise the designs of automated partitioning and a set of replica placement algorithms, ElasCass has to deal with a much more dynamic partition-node mapping than

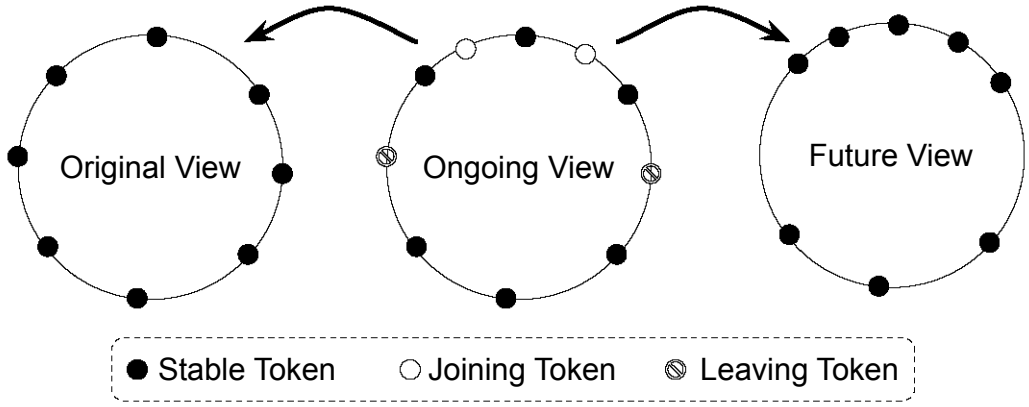


Figure 5.3: The ongoing view of key space in ElasCass

Apache Cassandra. This subsection presents the implementation of token management scheme for maintaining the partition-node mapping.

An Ongoing View of Key Space

To implement automated partitioning described in Subsection 4.2.1 (page 71–page 75), there is a challenge in associating the data files with a partition. That is, when certain partitions are being split or merged, different nodes have different views of the key space. To illustrate, the nodes that do not participate in the split/merge operation simply overlook the operation, while the participating nodes simultaneously deal with two versions of key ranges for each targeted partition: one is the original key range, and the other is the future key range as a result of the split/merge operation.

To avoid confusion between multiple versions of key ranges for the sake of automated partitioning, each node has an ongoing view of the key space. As shown in Figure 5.3, the ongoing view takes into account of the ongoing partitioning operations. To illustrate, during a split operation, the new token that divides the targeted partition is marked as a *joining* token; It is then marked as *stable* when the split operation is completed. Similarly, when a merge operation is in progress, the token to be removed is marked as a *leaving* token, and is actually removed when the merge operation succeeds. Hence, an original view of the key space is to ignore all joining tokens, but to retain all leaving tokens. In contrast, a future view is to add all joining tokens, but to remove all leaving tokens from the key space.

In terms of query processing, every node in the system uses the original view to serve key lookups, because the key space should not be altered until the split/merge operation

Multimap	Roles in queries	Purposes
Token-To-Node	Negative: write only; Positive: writable, readable.	Maintains the set of related nodes for each token in the original view of key space, making key lookups easier.
Node-To-Token	Negative: write only; Positive: writable, readable.	Maintain the set of tokens in the original view for each living node, making node-oriented statistics gathering easier.
Joining-Token	Always writable, readable	Maintain the set of nodes for each joining node, and help keep track of a partition-splitting operation.
Leaving-Token	Always writable, readable	Maintain the set of nodes for each leaving node, and help keep track of a partition-merging operation.

Table 5.2: The token-node multimaps in ElasCass

finally succeeds. However, from the perspective of the participating nodes, all the data files (i.e. SSTables) should be created based on the future view of the key space, which is the process of rebuilding replicas described in Subsection 4.2.1 (i.e. Figure 4.3 on page 75). Moreover, when a token is mapped to certain node, the token is assigned with a boolean value (i.e. either *positive* or *negative*) to indicate the responsibility of the node in serving queries, which has been discussed in Subsection 4.2.4 (Table 4.2 and Figure 4.7, page 89–page 93).

Storage of Token-Node Mappings

Figure 5.3 depicts three types of tokens, namely *Stable*, *Joining*, and *Leaving*, which yield different views of the key space for different purposes. Moreover, the mappings between tokens and nodes are many-to-many, and each token-node mapping requires a boolean value indicating the role of the node towards the related partition (Figure 4.7 on page 91). In addition, a mechanism to reconcile the conflicting mappings stored in different nodes is required, since the mapping information is shared without the presence of a centralised component.

ElasCass deals with the many-to-many relationship with a multimap, which maps a *key* to a set of *values*. It uses four multimaps for each key space to store different types of tokens. First, a *token-to-node* multimap maps a token in the original view to the nodes that own the token, wherein each node is bound with a boolean value indicating the node's responsibility. Second, a *node-to-token* multiple is the reverse map of the token-to-node

multimap, with the boolean value bound to the token. Last, a *joining-token* multimap and a *leaving-token* multimap store the mappings from a token to the related node for the joining and leaving tokens, respectively. The purposes and the roles of these mappings in query processing have been summarised in Table 5.2.

For the sake of conflict reconciliation, ElasCass uses a multikey map (i.e. from multiple keys to a single value) to record the type with a timestamp of update for each *token-node* entry, that is $\langle token, node \rangle \rightarrow (type, timestamp)$, wherein the *type* can be one of *Stable*, *Joining*, *Leaving*, or nonexistent. This timestamp is created by the node that initiates the update of the binding type, and cannot be modified by other nodes that store it, unless the type is altered in a later event (with a later timestamp). Furthermore, it is worth mentioning that these mappings are also stored in the system metadata table for durability, so that a temporary node failure can be self-recovered without resorting to message exchange.

Gossip Message for Token Update

ElasCass follows Cassandra to leverage the gossiping protocols for propagating the updates of token ownership. Cassandra only broadcasts the one-to-one mapping between the token and its coordinator node, so the content of a gossip message is relatively simple. In contrast, ElasCass maintains an ongoing view of key space per column family with a many-to-many mapping between tokens and nodes. Hence, a much richer gossip message format is required in ElasCass.

As shown in Table 5.3, a gossip message consists of two IDs (i.e. column family and node), one operation with its execution status, one boolean value for the token, and two timestamps. To illustrate, ColumnFamily ID indicates the related key space to update, while Node ID specifies the node to which the operation is applied. Next, the *add* and *remove* operations are associated with replica migration, while the *split* and *merge* operations deal with automated partitioning. Moreover, reporting the execution state of an operation helps each node to maintain the ongoing view of key space (Figure 5.3). In addition, each token is assigned with a positive or negative value to indicate different responsibilities of the node (Figure 4.7 on page 91).

Furthermore, there are two timestamps. One is the update timestamp, which is used to reconcile multiple conflicting token updates. Note that each token-node mapping is

Variables	**Possible Values**	**Notes or examples**
Column Family	ColumnFamily ID	Each column family has a separate key space.
Node	Node ID	The node to which the operation is applied.
Operation	Add	Assign a partition to a node.
	Remove	Remove a partition from a node.
	Split	Split a partition into two sub-partitions.
	Merge	Merge two partitions into one.
Execution state	In progress	E.g. a partition is *being* migrated to a node.
	Complete	E.g. a partition has *been* added to a node.
Token	Positive	A node is ready for query processing.
	Negative	A node is receiving a partition replica, but is not yet ready for query processing.
Update time	Timestamp	A timestamp indicating when the update was performed. It is used for versioning.
TTL	Timestamp	A time-to-live timestamp indicating when the message expires.

Table 5.3: Variables of a gossip message for token updates in ElasCass

assigned with a timestamp. Thus, only the update with a later timestamp will be applied. The other is the TTL (time-to-live) timestamp that sets an expiry time for a gossip message to spread. Each gossip message, when created, is assigned with a TTL timestamp, which equals the current time plus a TTL time period (the default is 30 seconds). Thus, if a node receives a gossip message at a time point later than the TTL timestamp, then it discards this message, which is otherwise forwarded to a set of randomly selected nodes. The maximum TTL (denoted as $MTTL$) represents the end-to-end gossip broadcast delay, and is important for controlling the timing of exchanging messages for replica migration between the source and destination nodes.

It is also worth mentioning how a node synchronises the token-node mappings, when it first joins the cluster or restores from the failure. For a new empty node, it will download the complete mapping information from a healthy node that is responsive and alive for a relatively long time. This new node will also store the mappings in its system metadata table, and updates the entries whenever it receives new gossip messages.

On the other hand, if it is an existing node that has recovered from a failure, then this node will reuse the mappings read from the system metadata table. In this case, it sends out gossip messages for each unsettled token-node mapping along with the update

Node Set	**Purposes**	**How to generate**
L set	Serve online reads and writes.	Retrieve node set from *Token-To-Node* given the token, and exclude all negative nodes.
K set	Maintain high availability.	Retrieve all nodes from *Token-To-Node* given the token.
Partitioning participants	Execute a partition operation.	Given the token, retrieve all nodes from *Token-To-Node* and *Joining-Token*. Then exclude all nodes retrieved from *Leaving-Token* with the same token.

Table 5.4: The set of eligible nodes for different operations in ElasCass

timestamp read from the table, wherein unsettled mappings include a joining or leaving token, or a token-node mapping bound with a negative value. On receiving such gossip messages, other living nodes will compare the update timestamp with their own records. A new gossip message containing a newer update will be sent out if a conflict is detected, so that the resurrected node can receive this newer update.

Choose Eligible Nodes during Token Update

Now that the data structure for storage and the message format (i.e. Table 5.3) for information exchange have been introduced, we discuss how to handle a token update event within each node. The challenge lies in efficiently providing the right set of nodes for different data storage and retrieval scenarios, because key lookups stand in the critical path of query processing.

Given the multimaps described in Table 5.2, each ElasCass node prepares different sets of nodes for different operations. First, to serve reads of certain partition, the node set is retrieved from the *token-to-node* multimap using the token that is associated with the partition. Only the nodes bound with a *positive* value are eligible for serving reads. Second, to serve writes of certain partition, it requires two sets of nodes. One is identical to the node set for reads, and is used to serve online writes (i.e. the L set in Table 4.2 on page 90). The other is the K set, wherein all the nodes mapped by the token (associated with the partition) are eligible, regardless of whether the token value is positive or negative. A write will be propagated to every node in the K set in the background for high availability. Both reads and writes use the original view of key space for key lookups. In contrast, to create new data files (i.e. SSTables) for a partitioning operation (i.e. split/merge), each participating node uses the future view instead, which has been depicted in Figure 4.3

(page 75). A summary of generating these node sets is presented in Table 5.4.

Note that an update of the token-node mapping can lead to the change of the node sets in Table 5.4. Such update is triggered by the event of replica reallocation or repartitioning, and is propagated in the form of gossip messages. On receiving a gossip message regarding a token update, a node validates the update timestamp and then applies it to the multimaps described as follows.

ElasCass uses an atomic *ReadWriteLock* to maintain a consistent view of the node sets during update. For the sake of efficiency, the node sets are generated once, and then cached in the query engine. This is done to avoid computing the node sets at each key lookup. When a token update is required, the node acquires a write lock for all the multimaps (Table 5.2), and then adds or removes the token-node mapping to the multimaps based on the type of the event. After the update, it invalidates the node sets cached in the query engine, recalculates the eligible node sets, and finally releases the write lock. During the time period when the write lock is held, no other process in the same node can obtain a write or read lock to access the token-node multimaps. Hence, a consistent view of the node sets is guaranteed.

5.3.2 Data Storage

This subsection presents the implementation of storing data objects into separate files based on the change in token-node mappings. First, it describes the generation of partition-specific SSTables to store incoming data objects. Second, it proposes an SSTable compaction algorithm that incorporates the process of rebuilding partition replicas into the routine of system garbage collection. Then, it presents the failover of an interrupted compaction. Next, it is followed by the discussion of selecting the right SSTables for serving reads when a compaction is in progress. Last but not least, it also describes the fulfilment of the prerequisites for executing a replica rebuilding task for the sake of automated partitioning.

Partition-Specific SSTables

As discussed, data migration is more efficient when the data objects are stored in separate files based on the partition to which they belong. ElasCass follows Cassandra to use the file structure called SSTable. As shown in Figure 5.2 (page 101), the SSTables are created as

a result of flushing the memtable onto the disk. In Cassandra, each memtable is converted into one SSTable, regardless of the partition to which each data object belongs.

In contrast, ElasCass needs partition-specific SSTables, wherein each SSTable only stores the data objects belonging to the partition with which it is associated. As in the discussion of Figure 5.3 (page 106), each ElasCass node uses its future view of the key space to determine the key range of the partition for each resulting SSTable.

ElasCass also partitions the key space using consistent hashing, wherein every two adjacent tokens define a partition. To convert the memtable into multiple SSTables, the data objects in the memtable are sorted by the keys, and the sorted list is segmented into multiple sublists by the tokens of the future view. Thus, each sublist of data objects are flushed onto the disk as one new SSTable. The key range of the partition with which this SSTable is associated, is defined by the two adjacent tokens that segment the sublist. In addition, these two tokens are written into the metadata of the SSTable, so that any node that loads this SSTable can perceive the associated key range.

Compared to Cassandra, given the same threshold for flushing the memtable, the size of the SSTable generated in ElasCass is much smaller, since each memtable is converted into multiple SSTables rather than one. It results in higher overheads in data retrieval, since there are more SSTables to visit. There are two remedies. First, ElasCass uses a higher threshold for memtable flushing, which requires higher memory consumption. Second, ElasCass adopts a levelled compaction strategy that prefers to merge small-size SSTables within the same partition. A complete SSTable compaction strategy is presented as follows.

SSTable Compaction Strategy

As in Bigtable (Chang et al., 2006), the immutable nature of SSTable requires a garbage collection (GC) process to regularly merge multiple SSTables into a large SSTable. This process is called the *compaction*, and has two goals. First, it reduces the total number of SSTables, which is especially necessary in ElasCass, wherein each memtable is converted into multiple small-size SSTables. Second, it allows the system to reclaim resources used by obsolete data and to ensure that deleted data disappears in a timely fashion.

In ElasCass, this GC process is applied not only to the merge of SSTables within the same partition, but also to the execution of rebuilding replicas when a repartitioning is

Algorithm 5 Generate SSTable compaction tasks during garbage collection

```
1: {P_i}_{i=1}^{q} ← getLocalPartitions()          ▷ Hint: the partition list is already sorted
2: for i = 1 → q do
3:     {f_j}_{j=1}^{m} ← getSSTables(P_i)
4:     size ← bytesOnDisk({f_j}_{j=1}^{m})
5:     if size ≥ Θ_max then
6:         submitSplitCompaction(P_i)                    ▷ To split the partition P_i
7:     else if size < Θ_min then
8:         P_next ← getSuccessor(P_i)
9:         if isLocalPartition(P_next) then          ▷ Try merge if both partitions are local
10:            {f_k}_{k=1}^{n} ← getSSTables(P_next)
11:            nextSize ← bytesOnDisk({f_k}_{k=1}^{n})
12:            if size + nextSize < Θ_min then
13:                submitMergeCompaction(P_i, P_next)    ▷ To merge P_i and P_next
14:            end if
15:        end if
16:    else
17:        upperBound ← 0.618 * size          ▷ 0.618 and 0.191 can be any other values
18:        lowerBound ← minimum(0.191 * size, Θ_minSize)
19:        Θ_minCmpt ← 4                          ▷ Minimum #SSTables for compaction
20:        toCompact ← newEmptyList()
21:        for j = 1 → m do
22:            if f_j.size ≤ lowerBound then
23:                toCompact.add(f_j)                ▷ Always compact small SSTables
24:            else if f_j.size > upperBound then
25:                markLargeFile(P_i, f_j)           ▷ Always avoid a large SSTable
26:            end if
27:        end for
28:        if toCompact.size ≥ Θ_minCmpt then
29:            submitGCCompaction(toCompact)
30:        else
31:            toCompact.addAll({f_j}_{j=1}^{m})             ▷ Compact all SSTables
32:            toCompact.remove(getLargeFiles(P_i))          ▷ Except the large one
33:            if toCompact.size ≥ Θ_minCmpt then
34:                submitGCCompaction(toCompact)
35:            end if
36:        end if
37:    end if
38: end for
```

required. Hence, there are three types of compactions. The process that merges SSTables from the same partition is called the *GC compaction*, while the processes that split or merge partition replicas for the sake of automated partitioning are called the *split compaction* and *merge compaction*, respectively. In ElasCass, each column family runs an independent GC process, which regularly inspects the metadata of all the SSTables in the column family. Then, a list of compaction tasks are generated according to Algorithm 5.

As shown, the GC process groups the SSTables by partition (which is written in the metadata of each SSTable), and calculates the total size of SSTables in each partition. According to Equation 4.1 (page 72), it submits a *split compaction* (Line 6), if the size exceeds Θ_{max}. Alternatively, a *merge compaction* is submitted (Line 13), if the total size of the examined partition and its immediate predecessor partition is below Θ_{min}.

When neither of the partitioning operations are triggered, the GC process will attempt to compact the SSTables within each partition. Note that the cost of SSTable compaction depends on the total size of SSTables to be compacted. Therefore, Algorithm 5 classifies the SSTables based on their sizes, into three categories: i) *large*, if the size is above the *upperBound* (Line 17); ii) *small*, if the size is below the *lowerBound* (Line 18), wherein $\Theta_{minSize}$ statically defines the size of a small SSTable when the total size of a partition is relatively large; iii) *normal*, if the SSTable is neither *large* nor *small*. Note that the predefined values in Line 17 and Line 18, i.e. 0.618 and 0.191, are derived from the golden ratio (related to Fibonacci sequence), and can be replaced by any other numbers.

The GC compaction always attempts to merge small SSTables, since the number of SSTables can be effectively reduced. There are also two refinements to avoid excessive compaction. First, the large SSTable will not be compacted unless it is explicitly required, because it already contains the majority of data of one partition. Second, a GC compaction will not be submitted if there are no enough SSTables to be merged ($\Theta_{minCmpt}$, Line 28 and 33). Also, when the number of small SSTables is less than $\Theta_{minCmpt}$, the *normal* SSTables are also included for compaction (Line 31), so that all the SSTables, except the large SSTable if there is any, can be merged into one SSTable.

Compaction Execution and Roll Back

A compaction task is submitted in Algorithm 5. A coordinator must be elected in the case of a split/merge compaction. The coordinator fulfils the prerequisites of partitioning, and then schedules a simultaneous compaction for all the participating nodes. The discussion regarding the prerequisites is appended to the end of this subsection on page 117.

In any case, once a compaction is set in motion, the memtable of the related column family is forced to flush based on the future view of key space, so that the latest writes are also compacted. Then, the targeted SSTables are marked as *compacting* in an atomic operation, such that the SSTables being compacted will not be submitted to multiple

concurrent compaction tasks. It also means that, if there exists one SSTable that is already marked as *compacting* by another thread, then this atomic operation of marking will fail, and the compaction task is canceled. Otherwise, if the targeted SSTables are successfully marked, then the node scans all the targeted SSTables simultaneously, and rewrites the data objects in the ascending order of their keys into one or two new SSTables, depending on the type of compaction (i.e. merge or split). At the end of compaction, the new SSTables are loaded into the system (e.g., build indices, cache Bloom filters, etc.), while the targeted SSTables are deleted from the disk.

A compaction task can be called off and rolled back at any time of the execution. This is necessary, especially when rebuilding replicas for a distributed split/merge operation. As depicted in Figure 4.5 (page 79), the partitioning operation should be aborted whenever there are more than one node failures. There are two scenarios to roll back from failure: one is for the well-functioning node that is called off by an abortion notification, and the other is in the case of a failed node. The challenge of rolling back a partitioning is to reconcile the conflicts between an original view and a future view of the key space (Figure 5.3 on page 106). The following discussion is focused on the split/merge compactions, but a GC compaction can also be rolled back similarly.

The first scenario is when a well-functioning node is required to abort the related compaction operation. When the node receives an abortion notification, it immediately terminates the compaction process and unmarks the related token from *joining*/*leaving*. Note that right before the compaction starts, the latest writes in the memtable have been converted into several new SSTables using the future view of the key space. The node has recorded these new SSTables for the sake of roll-back. Hence, these recorded SSTables are re-compacted if their associated partitions are inconsistent with the original view of the key space.

On the other hand, for a node that has failed during a compaction, it can be re-bootstrapped by loading the data from its own disk. Note that each node records the partition-node mappings in a system table on disk for faster recovery. When the node comes back to life, it recalls the previous view of key space, and it also attempts to synchronise the current view of key space as discussed on page 109. Next, the node examines the current status of the key space. If the previous split/merge operation has been canceled, then the resurrected node reuses those SSTables that are marked as *compacting*, and

also re-compacts those SSTables already built for the future partition. Otherwise, if the operation has succeeded without the presence of the resurrected node, then the node has to give up the ownership of the original partition and removes the related SSTables. In any case, the latest writes will be propagated to this node with the help of hinted handoff. Finally, the node is resurrected with the consistent ring information and data objects.

In both scenarios, the node will eventually delete those SSTables that are half-way built due to the interruption of failure, since they are unreadable. This roll-back process is efficient, because it is only involved with re-compacting several new SSTables converted from the memtable, the total size of which is small (i.e. between a few kilobytes and tens of megabytes). Such re-compaction (hence the roll-back) can be completed within one minute.

Data Retrieval during Compaction

For KVSs that support only GC compaction, data retrieval is straightforward. Since the SSTables marked as *compacting* and the resulting SSTable are associated with the same key range, the data can always be retrieved from all the SSTables available on disk. In contrast, ElasCass also support split and merge compactions, in which SSTables of the same partition can be associated with two different key ranges. That is, one set of SSTables fall into the original key range, i.e. before split/merge, while the other set of SSTables store data for the future key range, i.e. after split/merge. It is non-trivial to choose the right set of SSTables when a split/merge compaction is in progress.

As discussed previously, for the sake of rolling back a compaction, a node will mark the SSTables that are created from the memtable after the compaction has started. These SSTables fall into the future key range, but they also store data objects that belong to the original key range. Hence, to serve read operations, the key lookups will be based on the original view of the key space. But within each node, the data will be retrieved from two sets of SSTables: one set containing the SSTables belonging to the original key range, and the other set containing the SSTables that belong to the future key range and are marked for roll-back. Note that the SSTables created from the compaction will not be visited before the whole compaction task is completed. This is because the compacted SSTables are just extra copies of the data that is already stored in the two sets of SSTables.

In addition, at the end of a split/merge compaction, after the partitioning operation is

Algorithm 6 Calculate a splitting token for a list of SSTables in one partition

Require: $\{f_i\}_{i=1}^{k}$ ▷ A list of SSTables, each f_i provides *bytes*, *minKey*, and *maxKey*
1: $minToken \leftarrow getMinimumKey(\{f_i\}_{i=1}^{k})$ ▷ The minimum key of all the SSTables
2: $distanceList \leftarrow newEmptyList()$
3: **for** $i = 1 \rightarrow k$ **do**
4: $\quad distance \leftarrow getMiddleToken(f_i.minKey, f_i.maxKey) - minToken$
5: $\quad distanceList.add(distance, i)$
6: **end for**
7: $avgDistance \leftarrow (\sum_{i=1}^{k} distanceList.get(i) * f_i.bytes)/(\sum_{i=1}^{k} f_i.bytes)$
8: $middleToken \leftarrow avgDistance + minToken$
9: **return** $middleToken$

announced successful by the coordinator (Figure 4.4 on page 76), the token-node mapping is updated via gossip messages. Thus, the future key range becomes the original key range. Hence, the compacted SSTables are put in use, while the SSTables marked as *compacting* can be safely deleted in the end, because all the data objects in the *compacting* SSTables have been written into the compacted SSTables.

Fulfilling Prerequisites for Split/Merge Compaction

This subsection has discussed how ElasCass leverages the process of garbage collection (GC) to trigger a partitioning operation based on Equation 4.1 (page 72), and to execute the operation of rebuilding replicas as in Figure 4.3 (page 75). However, there are implementations required to fulfil the prerequisites of automated partitioning.

As depicted in Figure 4.4 (page 76), the execution of a split/merge operation relies on the election-based coordination. As discussed, the participating nodes are required to gossip their ownership of the targeted partitions before the election, so that an updated partition-node mapping can be seen by the "voters". The Chubby-based election and the fault-tolerant coordination have been discussed in Subsection 4.2.2 (page 75–page 80). The focus here is on how the coordinator fulfils the prerequisites before the execution starts, described at Step 2 of the coordination (page 77).

In the case of a *split*, the challenge is to efficiently determine the splitting token that segments a partition into two equal-size sub-partitions. The problem is that, each partition can contain many SSTables, and it is not efficient to scan over all SSTable data files to pinpoint the data object with the key that splits the partition into two equal halves. To address this problem, an algorithm is proposed to calculate the midpoint for a list of SSTables, based on only three facts that each SSTable metadata has already provided:

number of bytes on disk, the minimum key, and the maximum key of all the data objects in this SSTable. As shown in Algorithm 6, the splitting token is calculated as the weighted average distance of the middle token of every SSTable (denoted as f_i) from the minimum key in the partition, wherein the weight is the data volume of each SSTable (i.e., $f_i.bytes$). The idea of this algorithm is similar to calculating the centre of mass for multiple objects in physics.

In the case of *merge*, the challenge is to deal with the scenario when the two targeted partitions are not stored on the same list of nodes. ElasCass allows the coordinator to reallocate replicas for the sake of merging partitions, with one condition: only if all the source and destination nodes are *idle*. In the implementation, an idle node is defined as having CPU usage below 30%, which can be any other threshold. To reallocate the replicas of two partitions (e.g. P_i and P_{i+1}) onto the same node list, the coordinator always chooses the nodes that serve the latter partition (i.e. P_{i+1}) as the destination. The reason is that, when P_i and P_{i+1} are merged, the resulting partition is associated with T_{i+1}, while T_i is removed. On the other hand, if there exists at least one participating node that is not idle, the coordinator will cancel this merge operation, with a notification suggesting that the participating nodes should not submit a merge for the same pair of partitions within a long period (e.g. 30 minutes).

5.3.3 Replica Reallocation

Now that the replicas have been confined into transferable units, this subsection presents the implementation regarding replica reallocation. One aspect of replica reallocation is a set of algorithms to determine the placement of replicas when a node is bootstrapped or decommissioned, or fails unexpectedly, as discussed in Subsection 4.2.3 (page 80–page 89). One focus of these algorithms is to achieve load balancing in terms of both workload and distribution of data. This relies on the gathering of load statistics. The other aspect of replica reallocation is related to data migration discussed in Subsection 4.2.4, wherein the focus is on ensuring online query processing based on token management and transfer throttling.

Load	Metric	Means to gather statistics
Workload of nodes	CPU utilisation	Linux *sar* command.
Data volume of nodes	Number of replicas	Gossip and cache the complete partition-node mapping.
Popularity of partition replicas	Hit count	Hit count = number of SSTable reads (by queries only) + number of objects written to create SSTables. Both numbers are collected via the *data tracker*.

Table 5.5: Gathering load statistics in ElasCass

Gathering Load Statistics

The replica placement algorithms rely heavily on three facts: the workload that each node undertakes, the data volume each node stores, and the popularity of each partition replica. However, these load statistics are not provided in Apache Cassandra. The implementation choices in gathering these statistics are summarised in Table 5.5, while the discussion is presented as follows.

As mentioned in Subsection 4.2.3 (**Rule 3** on page 81), the CPU utilisation of a node is used as the metric for the workload. Such choice is due to two reasons. First, the CPU utilisation of a node is strongly correlated to overall response time of queries it serves (Lim et al., 2010). Second, the CPU utilisation can be obtained from the operating system without instrumenting application code. In the implementation, ElasCass uses the Linux *sar* command to collect CPU statistics.

In regard to the data volume each node stores, ElasCass uses the number of partition replicas as an indication, because each replica has been confined into a bounded size through automated partitioning. Moreover, since every node gossips and caches the complete partition-node mapping for the sake of query routing, the information about number of replicas in certain node is known to every bootstrapped node.

In contrast, measuring the popularity of a partition replica is less straightforward. As described in Equation 4.3 (page 83), the exponential moving average of the hit count is used as the popularity metric. However, there are a few issues to consider for measuring the accurate hit count for a partition replica. First, the write operations are stored in the memtable, which does not incur I/Os until it is flushed. Second, a read operation (i.e. retrieving certain data object) may result in accessing multiple SSTables due to the false-positive matches of Bloom filter (Bloom, 1970). Third, a data object may be cached in

memory, which does not requires I/O for retrieval. Last but not least, SSTable compaction also introduces a large number of I/Os.

To deal with these issues, we define the hit count of a partition replica as the sum of: i) the total number of visits to the SSTables that are introduced by read operations; and ii) the total number of data objects in the newly created SSTables. Hence, a read that involves with multiple SSTable accesses is treated as multiple hit counts, while a read from the cache does not cost any hit count. In terms of writes, since SSTables are immutable (i.e. write once, read many), the hit count of creating a new SSTable should be estimated as the number of data objects written. In addition, according to this definition of hit count, the workload of SSTable compaction is counted as the number of data objects stored in the compacted SSTables, while the number of reads introduced by compaction is disregarded. This is to reduce the impact of compaction towards the popularity of the partition replica.

The hit count measurement has been implemented in a component called *data tracker*, which is responsible for providing an atomic reference view of SSTables for query processing. Hence, this component is able to keep track of every SSTable visit and every new SSTable created. In the implementation, each column family has one *data tracker*, which maintains a hash map that records and updates the hit count of each partition periodically, i.e. based on Equation 4.3.

Exchanging Messages

The token management scheme for ensuring data consistency during replica movement has been depicted in Figure 4.7 (page 91). In terms of implementation, the focus is on the propagation of token updates and the coordination between the source and destination nodes.

Furthermore, the TTL (time-to-live) timestamp sets an expiry time for a gossip message to spread. Each gossip message is assigned with a TTL timestamp when it is created, wherein the TTL timestamp equals the current time plus the maximum TTL time period, which is 30 seconds by default. Hence, if a node receives a gossip message at a time point later than the TTL timestamp, then this message is discarded (rather than forwarded). The maximum TTL time period (denoted as $MTTL$) represents the end-to-end gossip broadcast delay, and is important for controlling the timing of exchanging messages for

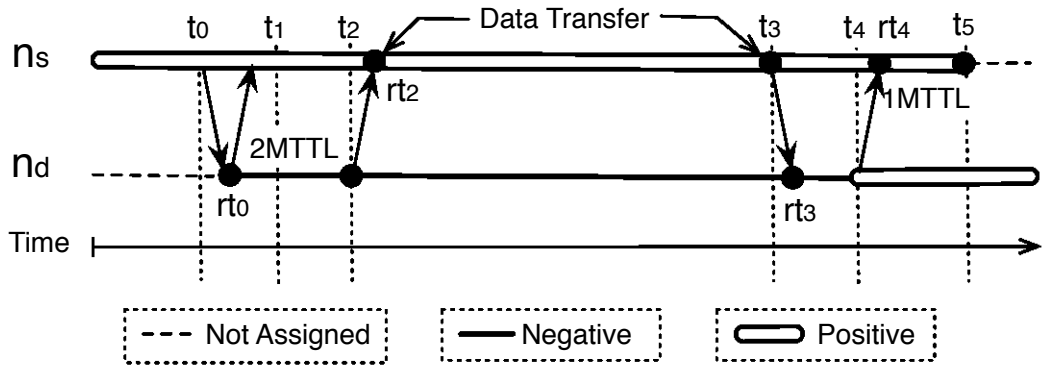


Figure 5.4: Real-time token management for replica migration

replica migration between the source and destination nodes.

Figure 5.4, extended from Figure 4.7, presents a closer look at the timing for migrating a replica from the source n_s to the destination n_d. As shown, when n_s initiates the migration at t_0, it immediately notifies n_d, which receives the the notification at rt_0 (rt means real time). Then, n_d assigns a negative token for the related replica to itself, and sends out a gossip message regarding this update. As discussed in the proof of data consistency on page 92, it is critical for n_d to await two time periods of $MTTL$ before it accepts the request for data transfer. Hence, $t_2 - rt_0 \geq 2 * MTTL$. Similarly, when n_d switches the token value to *positive* for serving read operations at t_4, it is also important for n_s to wait for one $MTTL$ before it abandons its ownership of the replica. Hence, $t_5 - rt_4 \geq MTTL$, wherein rt_4 is the time point when n_s receives the notification of token update sent by n_d.

Transferring Data

In ElasCass, the data is transferred in the form of data files (i.e. SSTables). The concern with data transfer is that it should not affect the performance of online query processing. There are two approaches to throttling the transfer bandwidth.

One option is to set up a TCP connection between two ElasCass nodes. The source node uses a daemon thread to determine the number of bytes sent at each time interval, such that the transfer speed in any time interval is no greater than a predefined maximum throughput for outbound transmission. Alternatively, ElasCass leverages the Linux *scp* command to perform file transfer in a more primitive fashion, which introduces lower overhead than application-level data transfer. The *scp* command provides the [$-l$ *limit*] parameter to limit the bandwidth used.

Table 5.6: Computational capacity of a single virtual machine in experiments

Property	**Value**
OS	Ubuntu 12.04, 3.2.0-29-virtual, x86_64
File system	Ext3
Instance Type	m1.large
Memory	7.5 GB
CPU	2 virtual cores with 2 EC2 Compute Units each
Storage	2 ephemeral storage with 420GB each
Disk I/O	High: 100,000 random read IOPS, 80,000 random write IOPS

The former approach has been used in Cassandra to transfer data at the granularity of data objects. In comparison, the latter is more suitable for transferring data in the form of files, but it is applicable to Unix/Linux based systems. Hence, ElasCass provides the implementation of both approaches, and the KVS administrator can designate the preferred approach in the configuration file.

5.4 Experiments and Results

5.4.1 Experimental Setup

This subsection presents the general setup for all the experiments in the following subsections. It describes the capacity of the experimental testbed, the configurations of the KVSs and the YCSB client, and the definitions of several performance metrics. The detailed setup for each specific experiment will be described in each subsection respectively.

The experiments were conducted on Amazon EC2. Each VM runs as one node of the KVS. All of the VM instances are based off a common Linux image. The computational capacity of a single VM is shown in Table 5.6. For performance reasons, the persistent data of the KVS was stored on the 400 GB ephemeral storage that comes along with each VM serving like a local disk, rather than on an Elastic Block Storage (EBS) volume (Amazon, 2007) that is accessed via network. This is consistent with known production deployments of Cassandra on EC2 (Cockroft, 2011). Expósito et al. (2013) also revealed that the use of ephemeral storage can provide better performance than EBS volumes, because the performance of EBS is deeply influenced by the network overhead and variability. The I/O performance of the ephemeral storage used in the experiments is categorised as “High”. According to Amazon, *High* I/O instances can deliver in excess of 100,000 random read I/O per second (IOPS) and as many as 80,000 random write IOPS.

The KVSs used in the experiments are ElasCass and Apache Cassandra (version 1.0.5) that uses the *split-move* approach discussed in Subsection 3.2.1 (page 57) for data partitioning and placement. An illustration of this approach is also given in Figure 5.1 on page 100. Hence, the experimental results of Cassandra are labeled as **split-move**. In addition, we did not compare ElasCass with other decentralised shared-nothing KVSs, such as Riak (Basho Tech., 2012), and Project Voldemort (2009), because they all use a similar split-move approach. Also, these KVSs use Berkeley DB or MySQL as the backend storage, which makes it less comparable with ElasCass or Cassandra.

In terms of configuration, both ElasCass and Cassandra follow the default setting in the release of Apache Cassandra, while the replication number is set as $K = 2$ for both systems, meaning that each data object should be copied onto two different nodes. In addition, ElasCass requires the settings of the maximum and minimum sizes of a partition as defined in Equation 4.1 on page 72. Although these thresholds vary in different experiments, the minimum size is always set as one half of the maximum, i.e. $\Theta_{min} = \Theta_{max}/2$. According to Equation 4.2 (page 73), the estimated average data volume per partition is equal to $0.625\Theta_{max}$.

The benchmark tools have been reviewed in Subsection 3.1.4 (page 51). The experiments used YCSB (version 0.1.4) (Cooper et al. 2010, GitHub.com 2010), because it has become an influential benchmark for KVSs (Cattell, 2011). The parameters configured are shown in Table 5.7. The data set is generated by the YCSB client in the *loading* section. The total size is approximately 100GB. The inserted keys are hashed with the 64-bit FNV function[1], so that the hotspot data is scattered onto many partitions. Both write-intensive and read-intensive workloads were generated using YCSB. Each workload was generated with two different request distributions, i.e. zipfian and hotspot. The consistency level (DataStax, 2012) is set as *ALL* for write operations, and *ONE* for read operations. This parameter specifies how many replicas must respond before a result is returned to the client. It tunes response time versus data accuracy, but does not affect the eventual consistency in Key-Value Stores.

In order to evaluate load balancing, an *imbalance* index I_L is defined to indicate the imbalance in load $\{L_i\}_{i=1}^n$ across a group of n nodes. Let $I_L = \sigma_L/\overline{L}$, where $\overline{L}$ is the average value of all the loads $\{L_i\}_{i=1}^n$, and σ_L is the standard deviation of $\{L_i\}_{i=1}^n$. This index

[1]Fowler-Noll-Vo is a non-cryptographic hash function created by Glenn Fowler, Landon Curt Noll, and Phong Vo.

Table 5.7: Parameters configured in YCSB

Property	**Value**
Records	size = 1KB, count = 100 million
Insert order	hashed with 64-bit FNV
Read/update ratio	50/50 for write-intensive, 95/5 for read-intensive
Request distribution	zipfian (constant = 0.99) hotspot (80% of requests targeting at 20% of data)
Operation count	varied
Thread count	varied
Consistency level	write: *ALL*; read: *ONE*

shows the proportion of the variation (or dispersion) from the average. A smaller value of I_L indicates better load balancing. We have evaluated the balancing of both the data volume and the query workload. Moreover, the average CPU utilisation is used to quantify the workload each node undertakes. The CPU usage is monitored periodically using the linux command "*sar -u 5 2*", which reports the average CPU usage every 10 seconds. Thus, we calculate the average CPU usage per node to indicate resource utilisation, and the imbalance index of the CPU usage of all the nodes to evaluate load balancing in the system.

The experiments were conducted in the following sequence. First, we evaluated the automated partitioning scheme, by partitioning the 100GB data using different values of Θ_{max} and Θ_{min} in ElasCass. Based on the experiment result, an appropriate setting of the thresholds was chosen for the next two experiments. Then, it is followed by the evaluation of node bootstrapping in both ElasCass and Apache Cassandra. At each system scale, a set of query workloads were launched against both KVSs, so as to evaluate their query performance in different scenarios. The experimental results of these three experiments are presented in the following subsections respectively.

5.4.2 Data partitioning

This experiment demonstrates how the maximum size Θ_{max} and the minimum size Θ_{min} (defined in Equation 4.1 on page 72) can affect the partitioning results with the automated partitioning strategy proposed in Subsection 4.2.1.

Test	Threshold	Est. avg. volume	Est. partitions	Initial Q	Initial replicas	Nodes
1	$\Theta_{max} = 1GB$, $\Theta_{min} = 0.5GB$	0.625 GB $= 1GB \times 0.625$	$160 = 100/0.625$	$32 = 160/5$	$64 = 32 \times 2$	**10**
2	$\Theta_{max} = 2GB$, $\Theta_{min} = 1GB$	1.25 GB	80	16	32	**10**
3	$\Theta_{max} = 4GB$, $\Theta_{min} = 2GB$	2.5 GB	40	8	16	**10**
4	$\Theta_{max} = 8GB$, $\Theta_{min} = 4GB$	5.0 GB	20	4	8	**2**
5	$\Theta_{max} = 16GB$, $\Theta_{min} = 8GB$	10 GB	10	2	4	**2**
6	$\Theta_{max} = 32GB$, $\Theta_{min} = 16GB$	20 GB	5	1	2	**2**

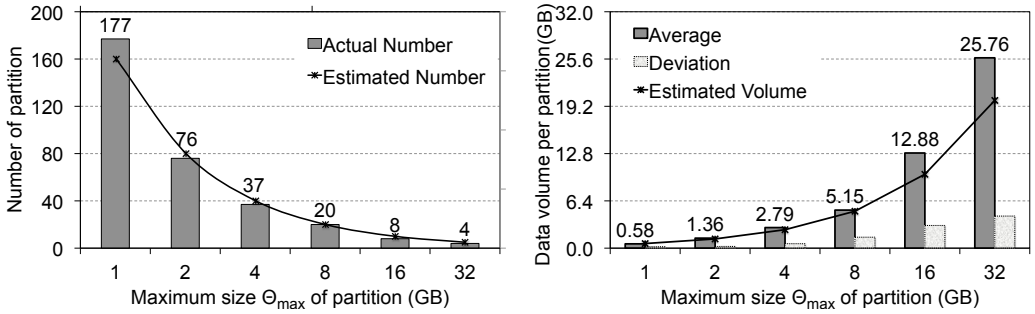
Table 5.8: The setup for evaluating automated partitioning

Experimental Setup

The YCSB client was configured with the parameters shown in Table 5.7. It generated the 100GB data set independently for each of the six tests in this experiment. In each test, the data was loaded into a cluster of empty nodes with different setups outlined in Table 5.8. As shown, the maximum size Θ_{max} increases exponentially, from 1GB to 32GB by the power of 2, while the minimum size is set as $\Theta_{min} = \Theta_{max}/2$.

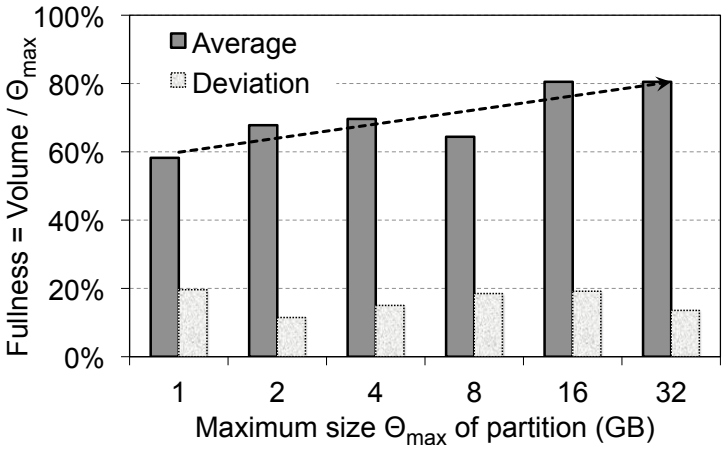
Note that ElasCass requires the setting of the initial number of partitions (i.e. Q), which segments the key space into Q equal-length key ranges. Given a total of 100GB data, the estimated number of resulting partitions is calculated from the estimated average volume. Then, Q is set to 1/5 of the estimated partition number, so that we can observe the execution of automated partitioning. Hence, Q decreases by the power of 2, starting from 32 when $\Theta_{max} = 1GB$, down to 1 when $\Theta_{max} = 32GB$.

Finally, the number of nodes employed in each test is determined based on the initial number of replicas in the cluster. Note that the replication number is $K = 2$ for all the tests. From Test 1 to Test 3, ten empty nodes were employed, since there are at least 16 initial replicas. In contrast, from Test 4 to Test 6, there are less than ten initial replicas, so two empty nodes were used instead. The replicas were evenly and randomly distributed across the nodes.



(a) Number of partitions

(b) Average data volume per partition



(c) Fullness of partitions

Figure 5.5: Statistics of partitioning 100GB data under different thresholds

Statistics of Automated Partitioning

Six tests were run based on the setup in Table 5.8, and Figure 5.5 presents a summary of the partitioning results. The total number of partitions generated is shown in Figure 5.5a, which also depicts the estimated number of partitions as a comparison. Note that the estimated number is calculated as 100GB (i.e. the total volume) divided by the estimated average volume per partition (Table 5.8). As shown, as the value of Θ_{max} increases, the actual number of partitions decreases inversely. Also, it matches the anticipated number under all thresholds, although it generates approximately 10% more partitions than estimated when $\Theta_{max} = 1GB$.

In contrast, Figure 5.5b shows the average volume of data stored in the partitions, which exhibits an increasing trend when the value of Θ_{max} grows. Moreover, the data volume in the resulting partitions agrees with the estimated volume when $\Theta_{max} \leq 8GB$, but is about one quarter higher when $\Theta_{max} = 16GB$ or $\Theta_{max} = 32GB$. The standard deviation of data volume also grows linearly with Θ_{max}. The clearer trends of the average

and the deviation are presented in Figure 5.5c.

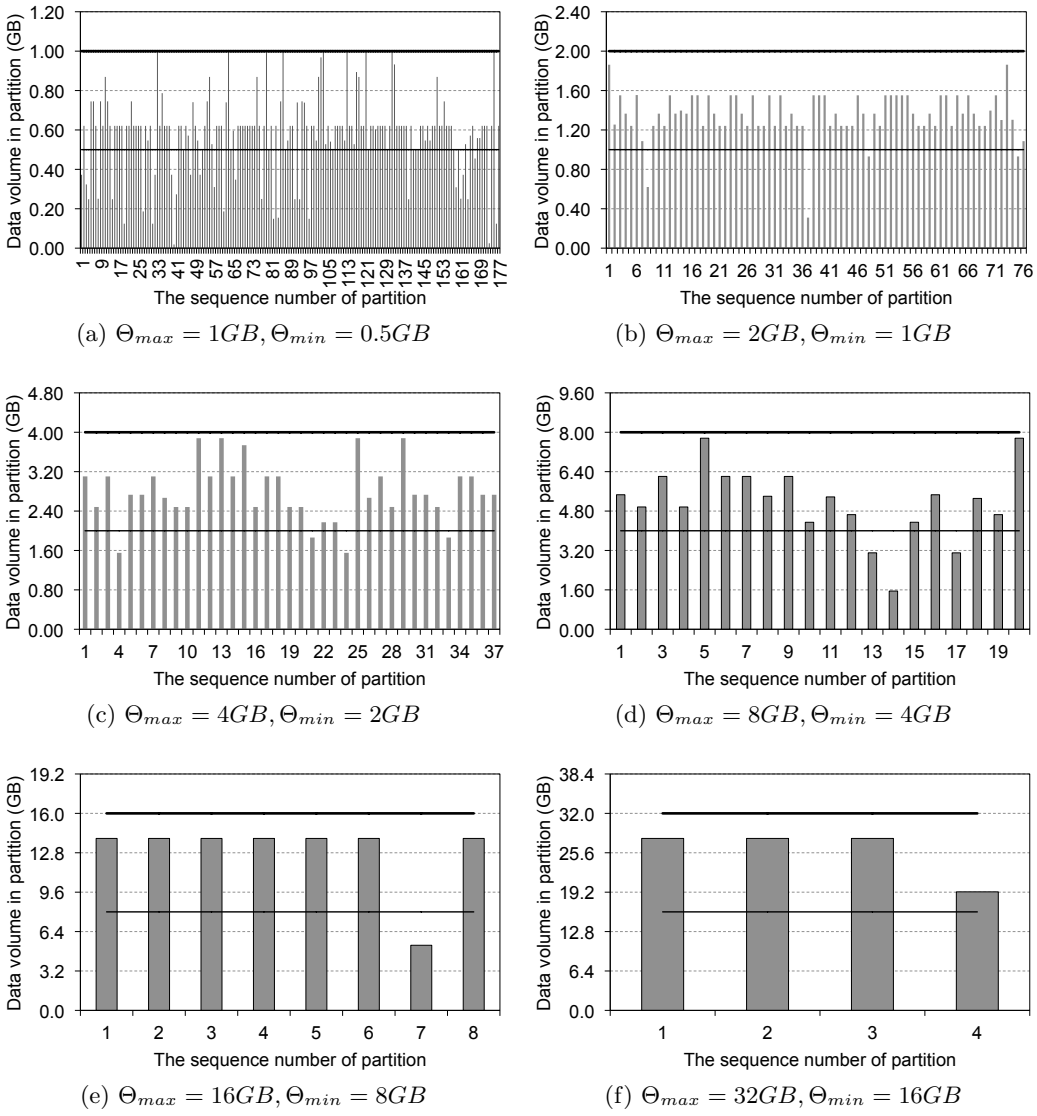
In Figure 5.5c, we use the term "fullness" to compare data volumes shown in Figure 5.5b. The value of fullness is calculated as the data volume of a partition divided by Θ_{max}. In other words, the fullness indicates how full the partition is before it reaches the maximum capacity. As can be seen, when $\Theta_{max} \leq 8GB$, the average fullness of the partition ranges between 60% and 70% when 100GB of data is inserted. This result is coherent with our estimation using Equation 4.2 (page 73) that the average volume is 62.5% of Θ_{max}. This fullness increases to 80% when $\Theta_{max} \geq 16GB$. Moreover, the standard deviation of fullness fluctuates between 10% and 20% given different Θ_{max}. The results indicate that the data set is effectively segmented into a list of partitions that are of roughly equal sizes.

In addition, there is a trend that larger values of Θ_{max} tend to result in greater fullness. This is because smaller upper bounds increase the number of partitions, which, as a result, increases the frequency of partition splitting and the difficulty of partition merging. The reasons are given as follows. First, when a partition is split, the data volume of each resulting partition is only half of Θ_{max}, i.e. the fullness is 50%. Second, one prerequisite for merging two partitions is that they have to be located on the same set of nodes. But such prerequisite becomes more difficult to meet when the number of partitions increases, in which case the remaining sparse partitions pull down the average fullness.

A Fine-grained View of Partitioning

Figure 5.6 presents the data volume of every resulting partition for the six tests. Each subfigure represents one test, wherein the partitions are displayed in order based on their key ranges. As can be seen, none of the partition sizes exceeds its corresponding Θ_{max}, which means the *split* operation was successfully triggered and executed whenever a partition grew over Θ_{max}. Moreover, the total size of any two adjacent partitions is above the corresponding Θ_{min}, meaning that the *merge* operation was able to eliminate consecutive sparse partitions for a more balanced distribution of data over the partitions.

However, Test 1 exhibits one sparse partition that was not merged as expected. As shown in Figure 5.6a, the 40^{th} partition is extremely small, and the sizes of both the 39^{th} and the 41^{th} partitions are well below Θ_{min}. According to Equation 4.1 on page 72), the 40^{th} partition should be merged with its adjacent partition. We analysed the system log,



(a) $\Theta_{max} = 1GB, \Theta_{min} = 0.5GB$

(b) $\Theta_{max} = 2GB, \Theta_{min} = 1GB$

(c) $\Theta_{max} = 4GB, \Theta_{min} = 2GB$

(d) $\Theta_{max} = 8GB, \Theta_{min} = 4GB$

(e) $\Theta_{max} = 16GB, \Theta_{min} = 8GB$

(f) $\Theta_{max} = 32GB, \Theta_{min} = 16GB$

Figure 5.6: A fine-grained view of partitioning 100GB data under different thresholds. The two horizontal lines in each subfigure represent the values of Θ_{max} and Θ_{min}.

and realised that, the nodes serving the 40^{th} partition did not host either the 39^{th} or the 41^{th} partition. A replica reallocation task (for the sake of merge) could not be executed either, because all the nodes were heavily loaded when consuming the 100GB data in bulk load. As discussed, allowing a small number of sparse partitions is more efficient than merging them aggressively. Nevertheless, it is the only exception out of the total 177 partitions in Test 1, so it still demonstrates the effectiveness of the *merge* operation.

In the remaining experiments, we have adopted the setting of Test 2, wherein Θ_{max}=2GB and Θ_{min}=1GB. There are two reasons. First, Figure 5.5c shows that Test 2 yields the least percentage of deviation in data volume, which can also be seen in Figure 5.6b, in which there are very few spikes. Second, it generates approximately 80 partitions, which

is a moderate number for a cluster of ten nodes.

5.4.3 Node Bootstrapping

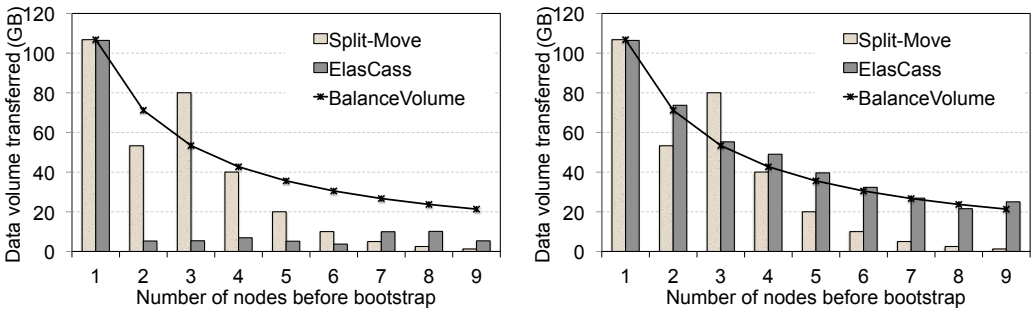
This experiment demonstrates the effects of bootstrapping nodes one after another, in a relatively short time, in both ElasCass and Cassandra that uses *split-move*. Thus, this subsection evaluates the efficiency of the proposed approach against the split-move approach for node bootstrapping.

Experimental Setup

In this experiment, both ElasCass and Apache Cassandra were initialised with one node. The 100GB data generated by the YCSB client was loaded on to the first node, with $\Theta_{max} = 2GB$ and $\Theta_{min} = 1GB$ in ElasCass, wherein the partitioning result is very close to Figure 5.6b (page 128). Since the replication number is configured as $K = 2$, when the second node was added, the 100GB data was automatically and completely replicated to the second node. From two nodes onwards, one empty node was added at each time. The data was reallocated according to different strategies in these two systems.

During the whole process of node bootstrapping, both KVSs were subject to a read-intensive background workload that followed the hotspot distribution (Table 5.7). We controlled the background workload by tuning the number of threads in the YCSB client, such that the CPU usage of the most loaded node was less than 80%, while the average CPU usage of all the existing nodes fluctuated around 50%. Therefore, both systems were *moderately* loaded before a new node was added. Moreover, each time before a new node was initiated, we made sure that every existing node had been serving queries for at least 15 minutes as a normal member of the KVS, so that the data reallocation had been settled for the previously added node. This workload also warmed up the popular replicas, so that it was possible for the KVSs to select the appropriate replicas according to the popularity.

Several metrics are measured in this experiment. First, the bootstrap time, which is denoted as the time between the start of the KVS process and the point when the node is ready to serve queries. Second, the bootstrap volume, which the volume of data acquired by a node at bootstrap. Ideally, the i^{th} node should share $1/i$ of the total volume of data in the system ($BalanceVolume$). However, this is affected by the partitioning and placement strategies employed. Therefore, the imbalance index of data distribution across the nodes



(a) Bootstrap volume comparison (b) Data volume acquired by the new node

Figure 5.7: The volume of data transferred at bootstrap in different Key-Value Stores

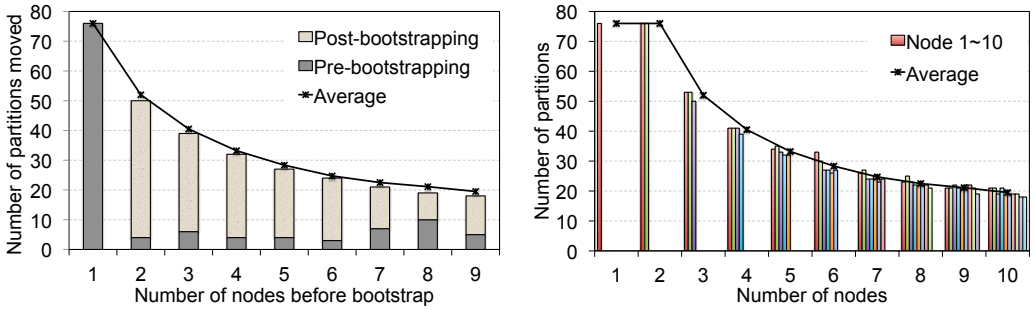
is also measured.

Data Movement at Bootstrap

Based on the setup, both KVSs scaled out from two nodes to ten nodes. Figure 5.7a depicts the bootstrap volume, while Figure 5.7b shows the total volume of data acquired by the new node. In both figures, the x-axis represents the number of nodes that are already in the system before each bootstrap.

As shown in Figure 5.7a, from two nodes onwards, the bootstrap volume in ElasCass fluctuates between 5GB and 10GB at all scales. In contrast, the volume of data transferred with the split-move approach drops exponentially as the system scales out. Note that ElasCass uses a two-phase bootstrapping strategy, and it deliberately limits the data volume transferred in the pre-bootstrapping phase (Algorithm 1, page 82), so as to achieve fast bootstrapping. However, Cassandra bootstraps a node in one step, so this result indicates that Cassandra does not balance the distribution of data, a fine-grained view of which will be presented in Figure 5.9a.

Figure 5.7b presents the total volume of data transferred at each bootstrap. The data volume transferred with the split-move approach is exactly the same as in Figure 5.7a, because a Cassandra node stops pulling data from its peers when it starts to serve queries. By comparison, a new node in ElasCass continues to take over more replicas during the post-bootstrapping phase, that is, after it begins to serve queries. Each new node uses Algorithm 2 (page 84) to determine the list of replicas to acquire. As can be seen, the total data volume transferred in this two-phase bootstrapping is roughly equal to *BalanceVolume* at each scale, which means that each new node in ElasCass obtained an equal share of data.



(a) Number of partitions selected at bootstrap (b) Partition Distribution

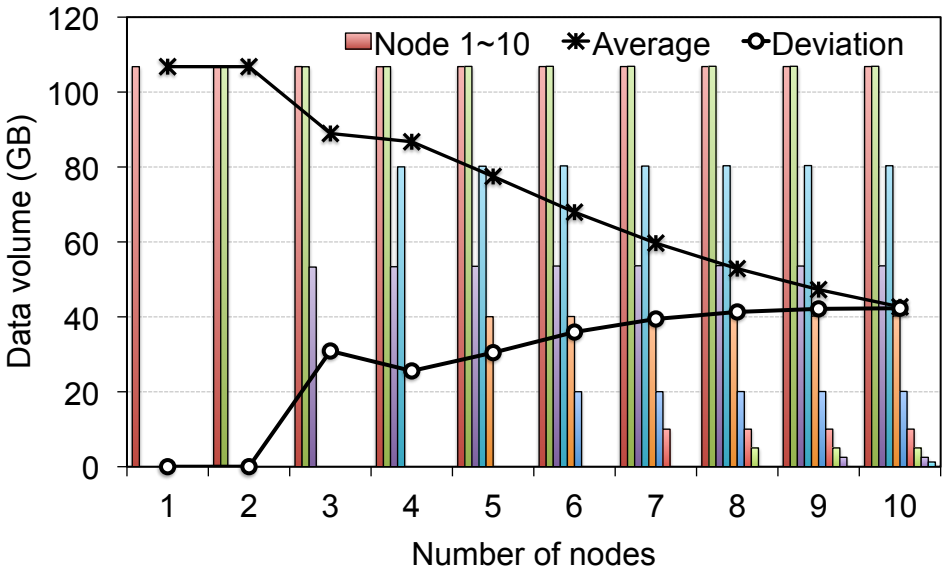
Figure 5.8: The change in number of partitions at bootstrap in ElasCass

Note that ElasCass maintains a balanced distribution of data by balancing the number of partition replicas in each node. Figure 5.8a depicts the number of partitions acquired by the bootstrapping node, and shows that, the total number of replicas assigned to a new node exactly matches the average number of replicas each node owns. Moreover, the changes in the number of partitions for each node is also presented. As shown in Figure 5.8b, there are ten groups of columns, wherein the $i^{th} group$ consists of i columns, each representing the data volume of a node at the scale of i nodes. It can be seen that each new node (represented by the rightmost column in each group), not only acquired an average number of partitions for itself, but also attempted to bring the partition number in every other node down to the average, by taking over partition replicas from them. Overall, ElasCass balances the number of partitions across the nodes at all scales.

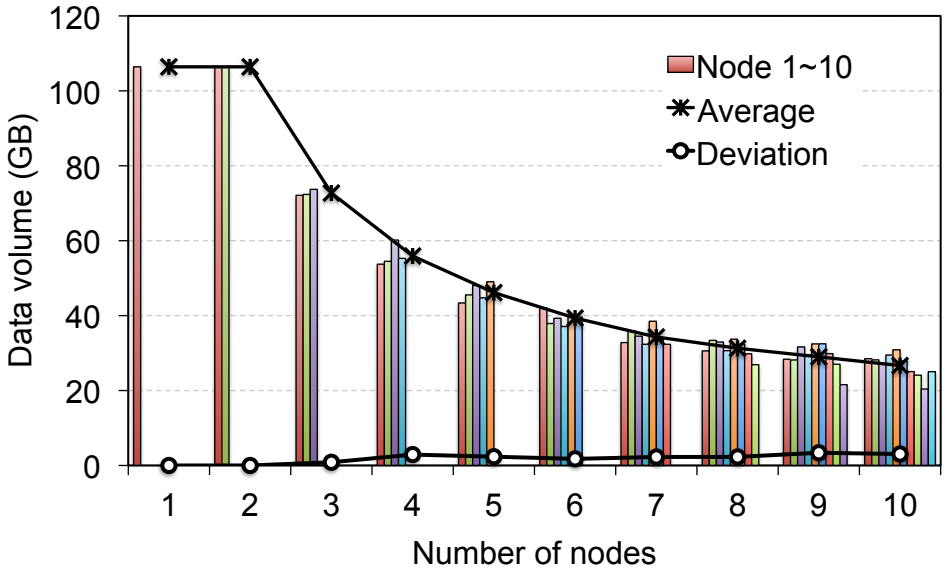
Data Distribution across Nodes

In order to analyse the bootstrap volume in both KVSs (Figure 5.7), a finer-grained view of data distribution in each node is presented in Figure 5.9. Both of its sub-figures follow the display of Figure 5.8b. That is, each system scale is represented as a group of columns, each showing the data volume in each node. The columns are also displayed in the order of node joining time. Thus, the i^{th} added node can be found at the i^{th} column in each group.

Figure 5.9a depicts the data distribution after each node bootstrapping with the split-move approach. As shown, after a new node was added the volume of data did not change in any node. For example, the second added node, represented by the second column in each group, stored over 100GB data on disk ever since it was bootstrapped. It means that, during the experiment, the data moved to the bootstrapping node, was not deleted at the



(a) Data distribution under different system scales with the split-move approach

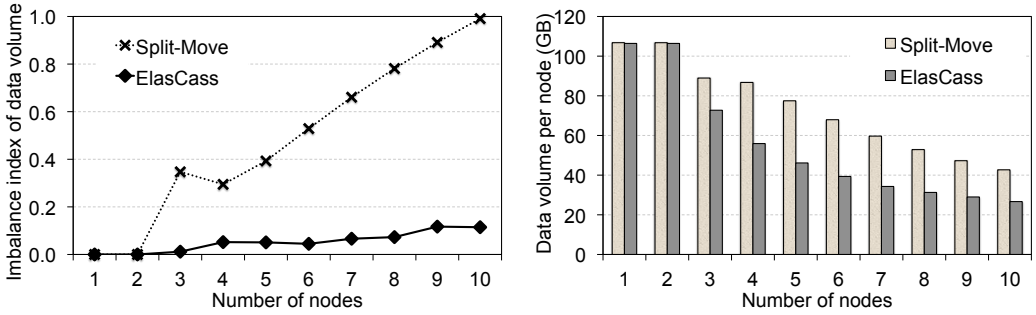


(b) Data distribution under different system scales in ElasCass

Figure 5.9: A fine-grained view of data distribution after node bootstrapping

source node. Such design is to avoid performing SSTable compaction (Subsection 5.3.2 on page 112) when the system is subject to heavy workload. Note that Cassandra relies on compaction to remove obsolete data from the immutable SSTables.

However, the obsolete data retained on disk had misled the subsequent new nodes. It is noticeable in Figure 5.9a that, from the 4^{th} node onwards, the data volume acquired by each new node (shown as the rightmost column in each group) drops exponentially by half at each scale, from 80GB at the 4^{th} node, down to merely 1.25GB when the 10^{th} node was bootstrapped. The reason is that, when a key range of data is moved to the new



(a) Imbalance index of data distribution (b) Data volume stored on disk at each scale

Figure 5.10: A summary of data distribution after node bootstrapping

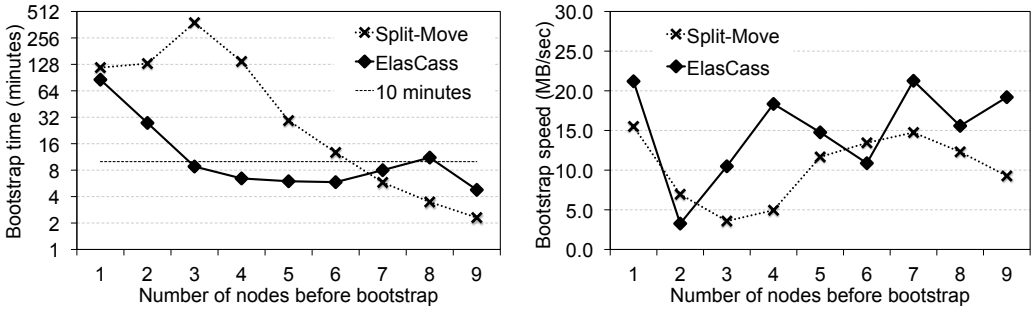
node, the persistent data is retained in (but no longer served by) the source node, until the SSTables are re-*compact*ed. However, since *compaction* is a heavyweight operation, it is rarely triggered when serving read-intensive workloads.

As a result, during the experiment, all the new nodes chose the same node (i.e. the node with the most data on disk) as the data source. Even worse, each time the chosen node had to offer half of its remaining key range, which was reduced by half each time a new node was added. This means that the load on the first three nodes was not reduced by introduction of the five new nodes, thereby defeating the purpose of elastically scaling the KVS. This is backed by the CPU utilisation presented in Figure 5.15a on page 139.

By comparison, Figure 5.9b demonstrates that ElasCass achieved a much more balanced distribution of data through node bootstrapping. The data distribution in ElasCass follows exactly the same pattern as the distribution of partitions shown in Figure 5.8b. This is because the data volume in each partition of ElasCass has been confined into a bounded range, shown in Figure 5.6b (page 128). In addition, unlike Cassandra that had to retain the deleted data on disk, ElasCass was able to remove the unwanted replicas effortlessly. The reason is that, each partition replica is stored in separate SSTables, which can be evicted from an ElasCass node as one unit, without any scanning or re-compaction.

Furthermore, Figure 5.10 summarises the data distribution details. Figure 5.10a compares the imbalance indices of data distribution for both ElasCass and the split-move approach. As described in Subsection 5.4.1, the imbalance index is used to indicate how imbalanced the targeted metric is over a set of nodes. Hence, a lower index means better balance. The imbalance index of data distribution is calculated as the standard deviation divided by the average data volume, both of which are shown in Figure 5.9a and 5.9b.

As shown in Figure 5.10a, the imbalance index of ElasCass remains at a low level at



(a) Bootstrap time (b) Data transfer bandwidth

Figure 5.11: Performance of node bootstrapping in different Key-Value Stores

all scales, which indicates that the data was evenly distributed in ElasCass. In contrast, in the split-move approach, the imbalance index soars to 1.0 within ten nodes, meaning that the split-move approach produced a very uneven distribution of data. This is also backed up by Figure 5.9a. At the scale of ten nodes, there are three nodes, each hosting 80GB of data or more. In the mean while, other five nodes serve less than 10GB of data on average.

In addition, Figure 5.10b shows the average volume of data stored on disk in all the nodes. The split-move approach occupies more storage because the source node that offers data at bootstrap tends to retain the invalid data on disk. By comparison, ElasCass is more efficient in storage space, since the partition replicas can be reallocated across nodes as one transferable unit.

Bootstrap Time and Speed

Now that the reallocation of data at bootstrap has been discussed, Figure 5.11 presents the performance of node bootstrapping in terms of bootstrap time and data transfer bandwidth, which is calculated as the data volume transferred (i.e. Figure 5.7a, page 130) divided by the bootstrap time. As mentioned, from two nodes onwards, both ElasCass and Apache Cassandra were subject to a moderate workload when bootstrapping a new node. Hence, the data transfer bandwidth was throttled by each KVS.

Figure 5.11a shows the bootstrap times in logarithmic scale for both KVSs with increasing number of nodes. ElasCass experienced a slow start at the scale of two nodes, but later on the bootstrap time fluctuates between five and ten minutes under different scales. By comparison, Cassandra, following the split-move approach, spent more than two hours in bootstrapping a node when there were four nodes or less in the system. From

the scale of four nodes onwards, the bootstrap time of Cassandra drops exponentially as the number of nodes grows, from 130 minutes to merely two minutes at the scale of nine nodes.

The result of Figure 5.11a can be explained by the bootstrap volume as shown in Figure 5.7a. At all scales from two nodes onwards, ElasCass managed to transfer less than 10GB of data constantly at bootstrap, as the remaining replicas were migrated in the post-bootstrapping phase. In contrast, with the split-move approach, the volume of data migrated decreases from over 80GB to merely 1GB. Specifically, from the scale of seven nodes onwards, the split-move approach did not migrate enough data as ElasCass did, thus requiring less time for bootstrapping. The penalty is that split-move suffered from load imbalance issue when serving queries, revealed in Subsection 5.4.4.

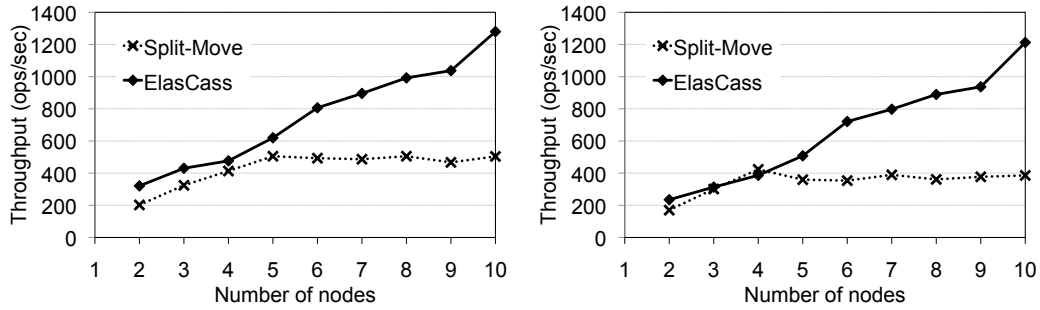
Moreover, Figure 5.11b depicts the data transfer bandwidth during node bootstrapping. Except for a slow start at two nodes, ElasCass managed to transfer the data at the speed fluctuating between 10MB/sec and 20MB/sec from three nodes onwards, while with the split-move approach, the data transfer bandwidth is relatively smaller, varying between 5MB/sec and 15MB/sec. The reason is that, in ElasCass, the data objects can be transferred in the form of files with the Linux *scp* command. Conversely, the split-move approach requires the source node to scan its data set to prepare the list of data objects, which are then reassembled into data files in the destination node. This process is more heavyweight than simple file transfer as in ElasCass.

5.4.4 Query Performance

This experiment compares the performance of query processing in ElasCass and Apache Cassandra that uses the split-move approach. As the two KVSs bootstrapped new nodes one after another (described in Subsection 5.4.3), a set of query workloads were launched against the KVSs at each system scale. This subsection evaluates the query performance and load balancing of both KVSs. Note that the experimental results of Cassandra are labeled as **split-move**.

Experimental Setup

This experiment is focussed on the improvement of workload throughput as the system scaled out. In order to measure the throughput at a steady state, we set an upper-bound



(a) Throughput under zipfian distribution (b) Throughput under hotspot distribution

Figure 5.12: The throughput of intensive writes with different distributions

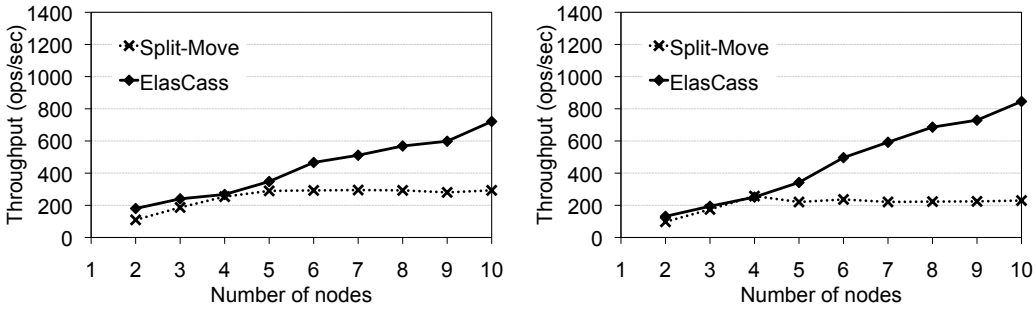
for the average read latency as 100 milliseconds (or ms), which can be considered as a service level objective (SLO) for certain web services. Before each test, we tuned the number of threads in the YCSB client, such that the average read latency is one-step below this bound. Based on this latency, we tuned the operation count (i.e. number of requests), so that the test can last long enough (at least 1000 seconds). Therefore, in all the tests presented, the average read latency is slightly less than 100 ms, and each run lasts at least 1000 seconds.

As mentioned in Subsection 5.4.1, there are two types of workloads, namely write-intensive and read-intensive, generated using YCSB. In a write-intensive workload, the read/update ratio is 50/50, while in a read-intensive workload, 95% of requests are reads. Moreover, each workload followed two different request distributions. In the *zipfian* distribution, the zipfian constant is set as 0.99, whilst in the *hotspot* distribution, 80% of the requests are targeting at 20% of the dataset.

Throughput at Different System Scales

Figure 5.12 depicts the throughputs of write-intensive queries in Cassandra (using split-move) and ElasCass against increasing number of nodes. As can be seen in Figure 5.12a, when the system is subject to write-intensive workload that follows the zipfian distribution, the throughput of Cassandra using split-move stops improving after adding the 5^{th} node, while in ElasCass, the throughput increases linearly with the number of nodes. When the workload following the hotspot distribution was applied, a very similar trend is shown in Figure 5.12b.

Moreover, Figure 5.13 shows the throughputs of read-intensive workloads for both KVSs. ElasCass continues to demonstrate better scalability than split-move when subject



(a) Throughput under zipfian distribution (b) Throughput under hotspot distribution

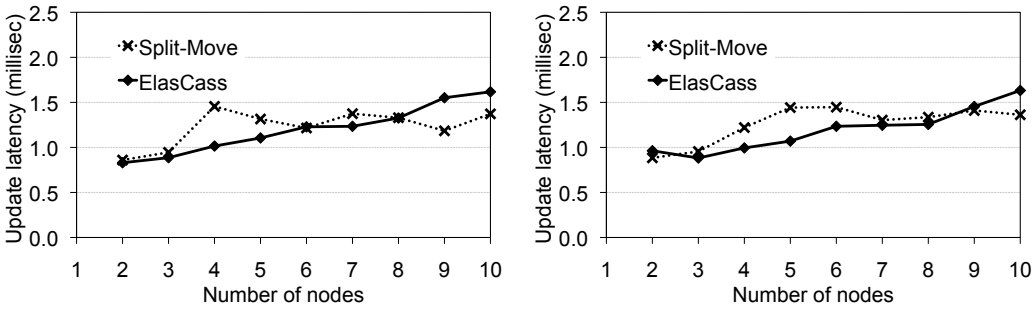
Figure 5.13: The throughput of intensive reads with different distributions

to read-intensive workloads under both zipfian and hotspot distributions. In addition, if we compare Figure 5.13 with 5.12, it can be seen that both systems have higher throughputs under write-intensive workloads than read-intensive. This is because write operations are buffered in memory and written in batch mode, while read operations require random disk I/Os, which are confined by the I/O performance.

It is clear that ElasCass outperformed Cassandra using split-move in terms of scalability by a large extent. The reason is due to the imbalanced distribution of data in the split-move approach (Figure 5.9a on page 132). Due to the lack of data moved to the new nodes, the split-move approach was not able to scale properly.

In practice, a *compaction* can be launched manually by system administrators before bootstrapping a node. Compaction will update the information of data volume on a node, so that a new node can choose source nodes more appropriately. However, this evaluation was designed to demonstrate how the KVS will behave without human intervention. Cassandra using split-move was not able to complete a compaction during the evaluation, which makes it unadaptable in the scenario where new nodes are added one after another in a relatively short time. In contrast, ElasCass is able to reallocate any partition replica as one transferable unit, since the data files are built in a partition-specific manner. Hence, the data volume on each ElasCass node is automatically updated effortlessly.

Furthermore, Figure 5.14 shows the update latency for both ElasCass and Apache Cassandra under write-intensive workloads. Note that the average read latency was tuned to be 100 ms to provide a steady state for the experiment, so it is not presented. As can be seen in Figure 5.14, the update latency fluctuates below 1.5 ms, which is two degree smaller than the read latency. This result suggests that both ElasCass and Apache Cassandra are optimised for writes.



(a) Update latency under zipfian distribution (b) Update latency under hotspot distribution

Figure 5.14: Update latency in write-intensive workloads under different distributions

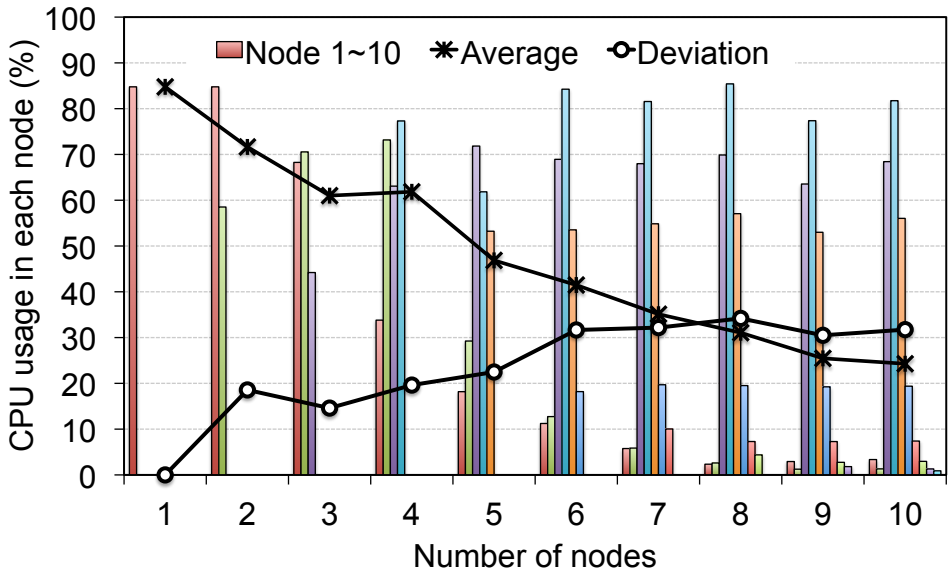
Comparison on Load Balancing

Figure 5.15 presents a finer-grained view of the CPU utilisation in each node during query processing. The layout of the two subfigures is consistent with Figure 5.9 (page 132). The workloads used in this figure are read-intensive queries that follow zipfian distribution. The CPU utilisations when serving other workload patterns show a similar pattern. Hence, they are not displayed, so as to avoid repetition.

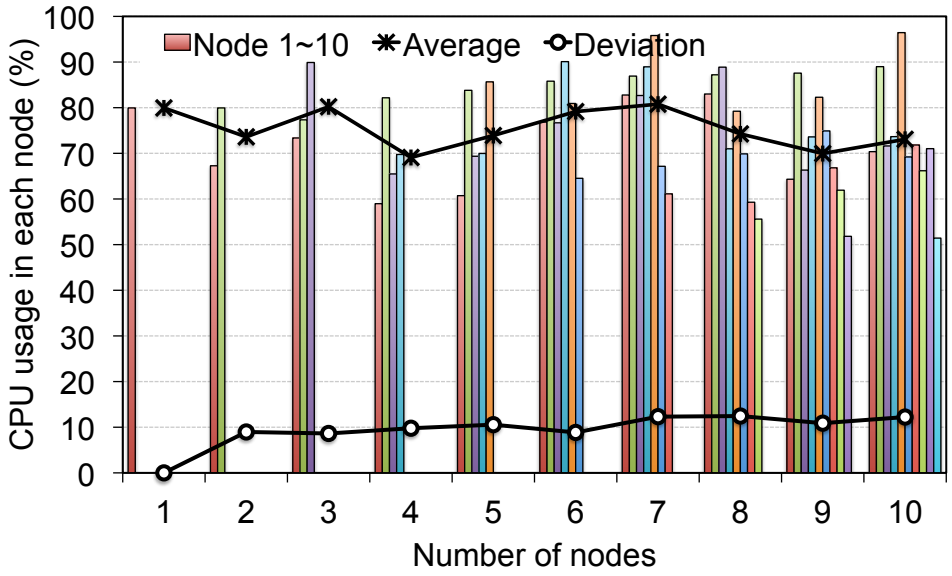
Figure 5.15a shows the CPU utilisations for each node in Cassandra using split-move. As can be seen, there are only three nodes at most with a CPU utilisation over 50% at each scale, while the other nodes are under-utilised, with a CPU usage below 20%. This unveils the reason why the throughput with the split-move approach stops improving in Figure 5.12 and 5.13, because the majority of the workload was shared by only three nodes, while the other nodes remained idle most of the time.

It is also worth mentioning that, the CPU utilisations of the first two nodes (represented by the first two columns), drops drastically from over 70% at the scale of three nodes, to below 10% from six nodes onwards. Apparently, the workloads were taken over by the 3^{rd}, 4^{th} and 5^{th} nodes, which consistently exhibit a high CPU utilisation at different scales. It can be inferred that the first two nodes had offered their data to all the other nodes. This is backed by a comparison between Figure 5.15a and Figure 5.9a. It shows that these three nodes hosted a total data volume of over 170GB, which occupies 85% of the 100GB data with the replication number $K = 2$. Conversely, while the first two nodes stored over 100GB each, their CPU utilisations were very low during query processing.

In contrast, Figure 5.15b depicts the CPU utilisation for each ElasCass node at different scales. As shown, every node in ElasCass has a CPU utilisation over 50%, with the average



(a) CPU utilisation of each node with the split-move approach

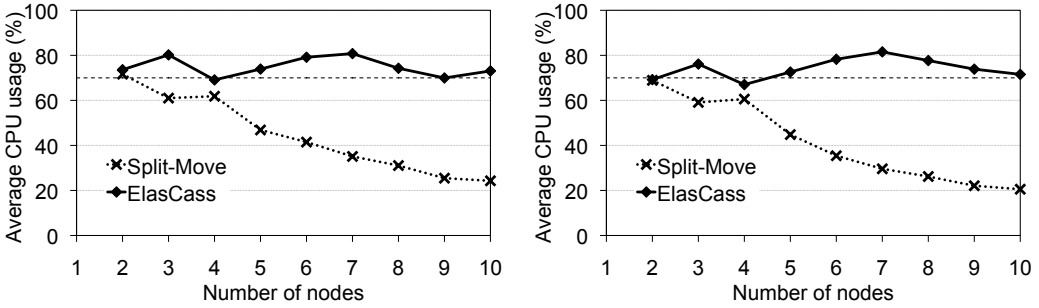


(b) CPU utilisation of each node in ElasCass

Figure 5.15: A fine-grained view of CPU utilisations under read-intensive workloads

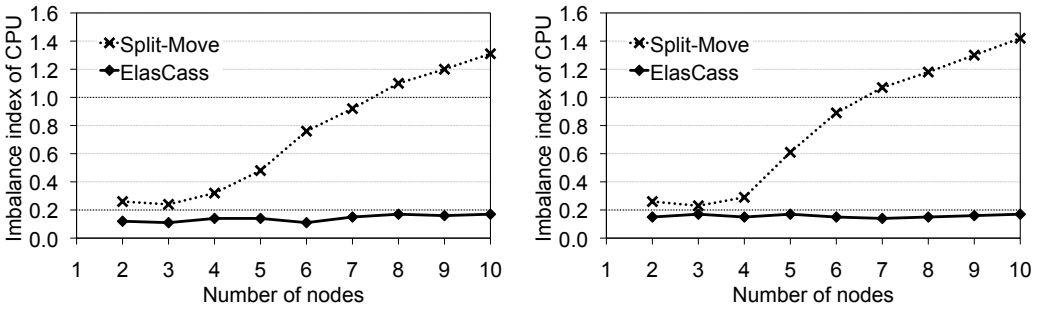
CPU usage being above 70%. This result indicates that all the nodes are full utilised in ElasCass. It also explains why ElasCass exhibited a higher scalability than Cassandra in Figure 5.12 and 5.13, since every node can devote its majority of computing capacity to query processing. Moreover, the standard deviation of the CPU utilisations fluctuates constantly at around 10%, meaning that the workloads are well balanced in ElasCass at different scales.

Furthermore, a summary of the CPU utilisations is also presented to compare the overall performance of both KVSs in load balancing. Figure 5.16a and 5.16b show the



(a) Using workloads under zipfian distribution (b) Using workloads under hotspot distribution

Figure 5.16: Average CPU utilisations of serving read-intensive workloads at varied scales



(a) Using workloads under zipfian distribution (b) Using workloads under hotspot distribution

Figure 5.17: Imbalance index of CPU utilisation when serving read-intensive workloads

average CPU usage of all the nodes during the tests under the read-intensive workloads, with zipfian and hotspot distribution, respectively. The results for the write-intensive workloads show a similar trend, so they are not presented to avoid repetition. As seen, the average CPU usage of ElasCass remains above 70% in both workload distributions. However, with the split-move approach, the CPU usage declines gradually as the system scaled out. The results indicate that ElasCass is able to fully utilise the provisioned computing resources at different scales for serving queries, while in Cassandra the newly added nodes were not efficiently incorporated into query processing.

Figure 5.17 presents the imbalance index of the CPU usage. With the split-move approach, the imbalance index climbs up as the system scales. In both subfigures, from the scale of eight nodes onwards, the index even goes beyond 1.0, which means that the standard deviation of the CPU usage is even greater than the average usage. The results indicate that some nodes are heavily loaded, while the others remain idle. The workload was not balanced with split-move. However, this index in ElasCass remains below 0.2 in all the tests. A small value of imbalance index indicates that the workload is well balanced in ElasCass.

5.4.5 Discussion

Three experiments have been presented in this section. Subsection 5.4.2 demonstrates that the data is evenly distributed across the partitions with the proposed automated partitioning. It can also be inferred that different settings of Θ_{max} (and Θ_{min}) can affect the system's performance. If Θ_{max} is too small, partitions are split very frequently, which increases the overhead of building replicas. In addition, small Θ_{max} results in a large number of partitions, which increases the complexity of partition reallocation. On the other hand, if Θ_{max} is too large, the resulting partitions will contain a large volume of data. Moving a large-size partition replica may end up in overwhelming the node that takes it over. Moreover, it takes substantially long time to reallocate a large-size replica, which is not efficient.

Next, Subsection 5.4.3 presents the evaluation of bootstrapping nodes one after another. The experimental results demonstrate that, as the system scales, ElasCass outperforms the split-move approach in node bootstrapping several ways. First, it manages to bootstrap an empty node within a relatively short time (i.e. ten minutes), by limiting the number of partition replicas transferred pre-bootstrapping. Second, it distributes the data more evenly amongst the nodes in the post-bootstrapping phase, as it partitions the key space into a finer grain. Third, it occupies less storage, since it can evict any partition replicas as one unit. Last but not least, it exhibits faster data transfer speed at bootstrap, because the data objects can be reallocated across nodes in the form of integrated files.

Furthermore, Subsection 5.4.4 compares the query performance in ElasCass and Cassandra that uses split-move. The experimental evaluation demonstrates that, due to the balanced distribution of data via node bootstrapping, ElasCass exhibits well balanced load across the nodes, fully utilisation of the computing resources and great scalability as the system scales. In contrast, Cassandra using split-move suffered from the polarised distribution of data, and ended up in serving queries with a small group of nodes, which results in a bounded throughput and a low resource utilisation.

Finally, we also discuss the limitation in this set of experiments. That is, the replica placement algorithms for node departure (i.e., decommissioning or failure) were not evaluated. This is because node decommissioning is treated as the reverse of node bootstrapping, and is executed only when there are redundant resources. Compared to node bootstrapping under substantial workloads, data reallocation with idle resources is relat-

ively trivial in node decommissioning. Moreover, nodes that fail unexpectedly are usually re-bootstrapped or replaced by new empty nodes provisioned from the IaaS Cloud, which is essentially node bootstrapping as evaluated in Subsection 5.4.3.

5.5 Chapter Summary

This chapter follows the design of a set of data distribution schemes for improving the efficiency of elasticity in decentralised shared-nothing KVSs. It falls into two parts. Firstly, it has discussed the implementation of the proposed schemes to present ElasCass, i.e. Elastic Cassandra, which is built on top of Cassandra (Apache, 2009) that follows the split-move strategy. To realise the proposed schemes, three core functionalities have been implemented. To begin with, a token management component is devised to maintain a dynamic partition-node binding for automated partitioning and flexible replica reallocation. Then, it is followed by a data storage component, which consolidates the data files into standalone transferable partition replicas. Finally, a replica reallocation component is implemented to execute the set of replica placement algorithms proposed for better balanced workload and distribution of data.

Secondly, this chapter has presented the experimental evaluations of ElasCass against Cassandra that uses the split-move approach. The evaluations were conducted on the Amazon EC2 public IaaS, using YCSB as the benchmark tool. There were three sets of experiments, focusing on data partitioning, node bootstrapping, and query performance, respectively. These experiments have demonstrated that, ElasCass was capable of incorporating new empty nodes consecutively in a relatively short time, with ideally balanced data distribution and much better balanced workload than Cassandra. As a result, ElasCass exhibited better resource utilisation in computing and storage, and outperformed Cassandra in scalability by a large extent under the biased workloads. We also demonstrated the capability of our automated partitioning scheme to confine each partition into a bounded size, without data skew or sparse partitions.

The positive results from this experiment encourage further exploration into enabling the elasticity in KVSs. The next chapter will investigate how the requirement of elasticity, i.e. allowing dynamic node addition and removal at runtime, will impact the performance of data durability in KVSs. As discussed in Subsection 3.2.2, data durability is affected by the strategy of data placement. The placement algorithms adopted by ElasCass essentially

fall into the category of random replication, which may result in higher probability of data loss in the face of simultaneous failure of multiple nodes (Cidon et al., 2013). Hence, the next chapter carries forward the automated partitioning and lightweight migration schemes in ElasCass, and presents a novel replica placement scheme called ElasticCopyset, to improve data durability over the random replication strategy, while still maintaining the efficiency of elasticity.

Chapter 6

Replica Placement for High Durability and Elasticity

The ninety miles is only half of a hundred-mile journey – the going is toughest towards the end of a journey.

– Zhan Guo Ce (戰國策)

The previous two chapters presented the design of a set of data distribution schemes that have been implemented to present ElasCass, an elastic KVS that efficiently deals with node addition and removal on demand. This chapter presents a novel replica placement scheme called *ElasticCopyset*, which aims to minimise the probability of data loss when facing simultaneous failures of multiple nodes, while still allowing dynamic node changes for the sake of elasticity that is already established in ElasCass.

This chapter begins with the motivation of data durability in the face of multiple node failures, and then revisits the concept of copyset and its use in reducing the probability of data loss. Next, it provides the mathematical definitions for the problem of data durability. Based on the problem definition, it presents the design of ElasticCopyset, and the experimental evaluations. Finally, this chapter concludes.

6.1 Introduction

Distributed Key-Value Stores (KVSs) (Chang et al. 2006, DeCandia et al. 2007, Cooper et al. 2008) have become a standard component for data management for applications

deployed on the IaaS Cloud. This thesis focuses on the elasticity of KVSs, which requires the nodes of a KVS to be efficiently added or removed at runtime.

Data in KVSs is organised in *storage units*, called variously as chunks, tablets, partitions or virtual nodes, which are replicated for improving fault tolerance and performance. A fundamental design choice in KVSs is the placement scheme for distributing the replicas of data storage units across the nodes. The previous two chapters have presented a set of replica placement algorithms that achieve well balanced workload and data distribution for node addition and removal.

One characteristic of these placement algorithms is that the replicas of a data storage unit (i.e. a partition in ElasCass) can be assigned to any node in the KVS. That is to say, the mappings between replicas and nodes are randomised. As introduced in Subsection 3.2.2, such a placement strategy is termed as **random replication**. This strategy provides sufficient flexibility for a KVS to distribute replicas across nodes, and thus benefits the dynamic addition and removal of nodes in a KVS. Therefore, random replication has been adopted by many KVSs including Bigtable (Chang et al., 2006), Dynamo (DeCandia et al., 2007), and Cassandra (Lakshman & Malik, 2010).

However, Cidon et al. (2013) demonstrated that, once the size of the system scales is beyond hundreds of nodes, random replication is nearly guaranteed to cause data loss when there are multiple nodes failing simultaneously. The remaining of this section discusses the motivation of protecting a KVS from simultaneous node failures, and then reviews the current-state replica placement strategies that reduce the probability of data loss at simultaneous node failures.

Simultaneous Node Failures on the Cloud

The IaaS Cloud typically provisions computing resources in the form of virtual machines (VMs), which are simulated by a hypervisor (Popek & Goldberg, 1974), such as Xen (Dragovic et al., 2003) or KVM (Kivity et al., 2007), that runs a *host* on one or more real machines. In this way, each VM runs as a *guest* process within the host machine, and each host machine supports multiple VM processes.

With the growing trend of Cloud computing, IaaS Cloud platforms are usually built from hundreds of thousands of inexpensive, low-end commodity components. However, at such a large scale, the failure of a single hardware component at any time becomes

the norm rather than an exception (Ghemawat et al., 2003). The failure of such single hardware component can lead to the malfunctioning or even failure of multiple processes (e.g. VMs) that are running on the corresponding physical machines (Vishwanath & Nagappan, 2010).

Therefore, hardware failure in the IaaS Cloud can cause the simultaneous failures of multiple VMs. For example, Raphael (2013) presented a list of outage events in public Cloud providers, including Amazon, Microsoft, and Google, for a period of merely six months. Worse still, as documented in practice by Yahoo! (Shvachko et al., 2010) and Facebook (Borthakur et al., 2011), in the event of multiple node failures, a non-negligible percentage of machines (0.5%-1%) cannot be restored even after the failure is recovered (Cidon et al., 2013).

This has posed a threat to data durability in KVSs that are deployed on the IaaS Cloud. When a non-negligible percentage of KVS nodes (each running on one VM) fail simultaneously, if all the replicas of certain storage unit are located in the nodes that have failed in the event, then this storage unit of data becomes unavailable. Hence, there is a need for a replica placement scheme that focuses on reducing the probability of data loss. A further discussion is presented as follows.

Replica Placement Strategies for Data Durability

As discussed in Subsection 3.2.2 (page 61), previous research efforts (Chun et al. 2006, Cidon et al. 2013) have concluded that the random replication strategy, which randomly distributes the replicas of data across all the nodes in a KVS, results in a high probability of data loss at multiple node failures. This will be explained later in Subsection 6.2.2 using formal definitions in mathematical terms.

Instead of using random replication, Cidon et al. (2013) proposed to confine the placement of replicas into certain predefined sets of nodes, called "copysets". That is, each storage unit of data is assigned to a copyset of nodes, each node storing one replica for this storage unit. In this way, this data will stay available unless all the nodes in this copyset have all failed. This copyset-based placement scheme builds on the insight that minimising the number of copysets can minimise the probability of data loss. As demonstrated in Figure 6.1, given the same percentage of nodes that fail, the number of copysets dominates the probability of data loss. This conclusion is also backed by Subsection 6.2.2

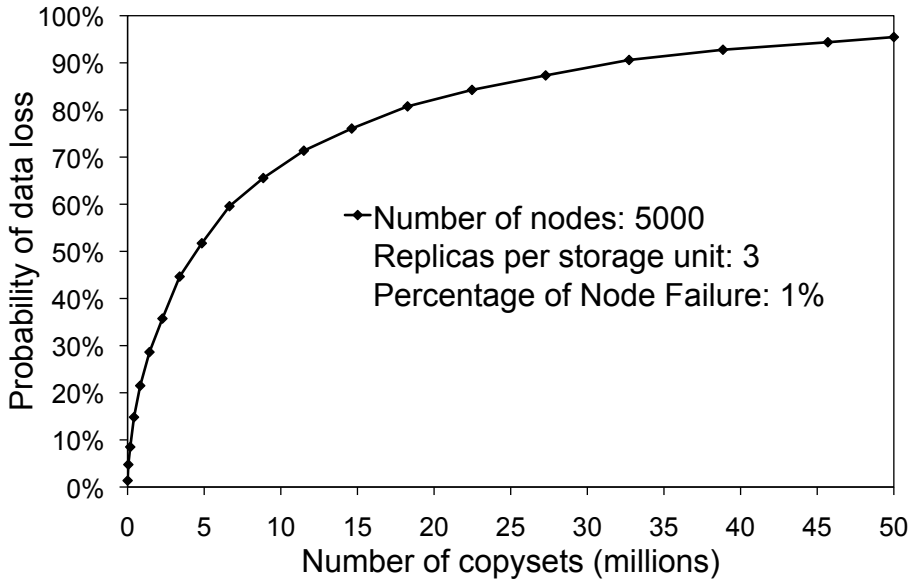


Figure 6.1: Probability of data loss versus number of copysets. Given the number of copysets required at each depicted point, distinct copysets were generated, based on permutation described in Copyset Replication (Cidon et al., 2013). The probability of data loss was calculated from simulating 10000 occurrences of simultaneous node failures, as depicted in Section 6.4.

with a proof in mathematical terms.

This copyset-based placement scheme embodies two trade-offs regarding the number of copysets. First, when one of the nodes in a copyset fails, the replacement node can only recover data from the remaining nodes in the same copyset. Thus, the speed of recovery is determined by the total number of nodes that share copysets with the failed node. Hence, a trade-off needs to be made between lower probability of data loss (by using fewer copysets) and faster recovery speed (by allocating each node into more copysets). Second, there is a trade-off involving the frequency and magnitude of data loss. In copyset-based schemes, data loss occurs only when all the nodes from the same copyset fail simultaneously (i.e. a copyset failure). The average amount of data loss in a single copyset failure is equal to the total amount of data divided by the number of copysets. That is, a smaller number of copysets results in higher amount of data loss in each failure event.

To deal with these two trade-offs, Cidon et al. (2013) introduced the concept of *scatter width*, which is defined as the number of nodes that store copies for each node's data. Using a high scatter width means that each node shares data with more nodes, which requires more copysets to be created. Thus, it improves the performance to incorporate or recover a node, but creates more opportunities of data loss under simultaneous failure of nodes. In contrast, using a low scatter width limits the number of nodes that each node

can share data with, thus requiring few copysets to be created. It reduces the probability that all the replicas of some data are lost when multiple nodes fail, with the trade-off of slow recovery time from independent node failures.

System administrators can make trade-offs by setting the preferred value of scatter width. Cidon et al. (2013) proposed a permutation algorithm that forms a minimised number of copysets based on a given value of scatter width, which is determined by system administrators. However, current-state copyset implementations (Cidon et al., 2012, 2013) do not consider the requirement of dynamic node addition and removal at runtime. This issue is discussed as follows.

Elasticity Requirement

The aim of this chapter is to carry forward the benefit of elasticity that has been established in the previous two chapters. However, the random replication adopted does not provide high data durability. In contrast, the current-state copyset-based placement scheme is able to minimise the probability of data loss at simultaneous node failures, but is not adaptive to dynamic node changes, discussed as follows.

In current implementation of copyset-based schemes (Cidon et al., 2012, 2013), when a new node is added to the system, it is not added to any existing copyset. Instead, a number of new copysets are created for this new node. The new copysets are populated using existing nodes randomly selected. Moreover, when an existing node is removed (or fails), its vacancies in the corresponding copysets are filled by an existing node that is also randomly selected. Therefore, node addition increases the total number of copysets, while node removal does not reduce the copyset number. As will be demonstrated in Figure 6.17 in Section 6.4, when the system is required to scale dynamically based on workload changes, the total number of copysets accumulates at each node addition, and eventually becomes sufficiently large such that the data durability downgrades severely.

Hence, there is a lack of placement scheme that simultaneously maintains a minimised probability of data loss at multiple node failures, while allowing the nodes to be dynamically added or removed at runtime. This chapter presents ElasticCopyset, a novel placement scheme that fills in this gap. The next section will provide definitions to formulate the problem of data loss at multiple node failures, while the design of ElasticCopyset will be elaborated in Section 6.3.

Table 6.1: Notational Conventions

Notation	Description
R	The replication number, i.e., #replicas of each storage unit
S	The scatter width
P_{ran}	The probability of data loss using random replication
P_{cs}	The probability of data loss using copyset replication
N	The number of nodes in the system
F	The number of nodes failed in a failure event
N_G	The number of nodes in each group
N_E	The number of extra nodes that cannot form a complete group
N_{su}	The number of storage units in the database
N_{cs}	The number of copysets in the system
C	The number of columns in the shuffle matrix
L	The number of rows in the shuffle matrix
a, b, c	The bucket of nodes to shuffle
m_a, m_b, m_c	The matrix of nodes after shuffle
x, y	The x, y coordinate of the shuffled matrix
i, j, k, n	Non-negative integers
r, s, t	Integers

6.2 Problem Definition

6.2.1 Parameter Definitions

We focus on the issue of data placement in distributed KVSs across a cluster of N data nodes. Typically, the data is horizontally partitioned, and stored as consolidated replicas. In this chapter, we define the *storage unit* as the basic unit for replication, which can refer to a key-value pair, a tablet or a data chunk in different data stores. Each storage unit is stored on a set of distinct data nodes, which is called the *copyset* for this storage unit. The notational conventions used in this paper are summarised in Table 6.1.

The number of distinct nodes in the copyset is defined by the *replication number* R, while the number of nodes that store copies for each node's data is defined as the *scatter width* S (Cidon et al., 2013). Figure 6.2 presents an example to illustrate. As shown, the storage unit E is replicated to the nodes $\{0, 3, 6\}$, so the copyset for E is $\{0, 3, 6\}$, wherein the replication number $R = 3$, since there are three nodes in total. In contrast, Node 0 shares its data with other six nodes, including the nodes $\{1, 2, 3, 4, 5, 6\}$, hence the scatter width for Node 0 is $S = 6$.

To reduce complexity, we use a unified replication number and scatter width in the system. That is, every storage unit is stored by R distinct nodes, which form one copyset, and every node shares its data with S other nodes. Therefore, it can be inferred that each

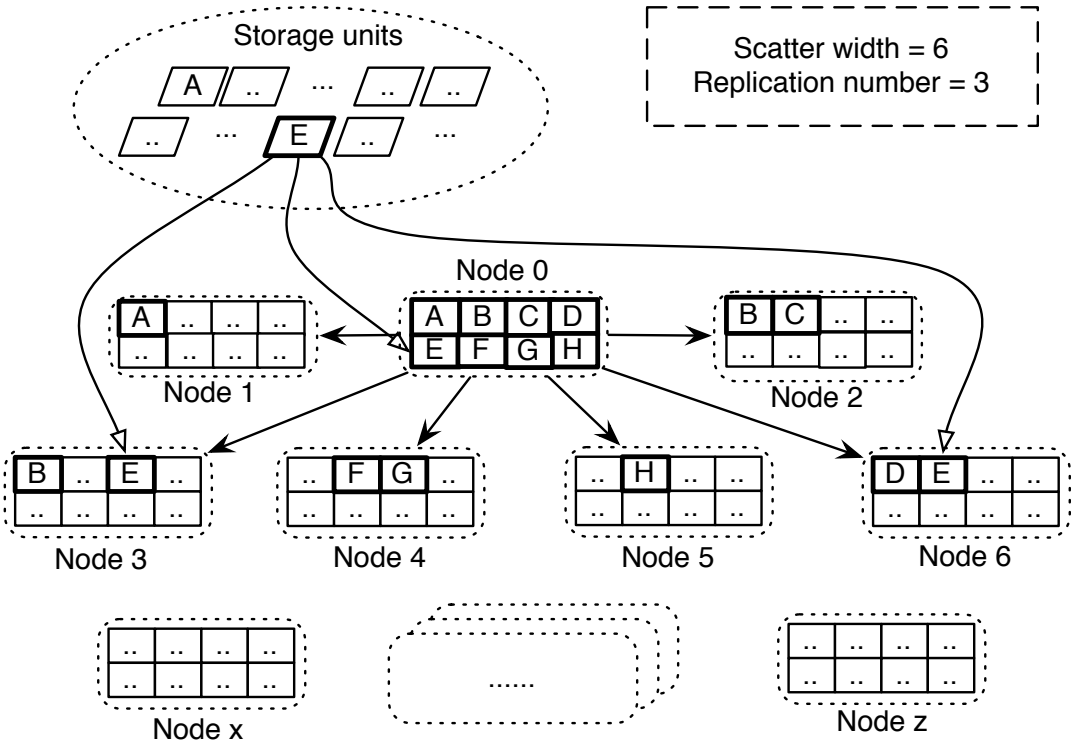


Figure 6.2: An illustration of scatter width S versus replication number R. In this example, each storage unit is replicated to $R = 3$ different nodes, while each node shares its data with $S = 6$ other nodes in the system.

node shares copysets with at least S other distinct nodes. Since each node has $(R-1)$ other nodes in each copyset, then each node should be assigned to at least $\frac{S}{R-1}$ copysets. Thus, $S \geq (R-1)$. The scatter width is determined by the system administrators, and is usually a multiple of $(R-1)$.

6.2.2 Relationship between Data Loss and Copysets

Probability of Data Loss

In a failure event, there are F nodes failed simultaneously. When $F > R$, there is a chance that all the R replicas of some storage unit will be lost in this failure event. The way that the copysets are formed can affect the probability of data loss substantially, discussed as follows.

In random replication, for each storage unit, R nodes are randomly chosen from all the nodes in a KVS to form a copyset for this storage unit. As discussed by Cidon et al. (2012), the probability of losing one single storage unit is $\frac{\binom{F}{R}}{\binom{N}{R}}$, wherein $\binom{N}{R}$ and $\binom{F}{R}$ denote the number of ways of picking R nodes unordered out of N and F nodes, respectively. Therefore, the probability of losing at least one storage unit in the scenario

of F simultaneous failures is given by Equation 6.1, wherein N_{su} is the number of storage units.

$$P_{ran} = 1 - \left(1 - \frac{\binom{F}{R}}{\binom{N}{R}}\right)^{N_{su}} \tag{6.1}$$

In contrast, in a copyset-based scheme, the copysets are generated independently from the data, and each copyset of nodes hosts multiple storage units in a copyset-based scheme. Therefore, the number of copysets, denoted as N_{cs}, is less than the number of storage units, i.e., $N_{cs} < N_{su}$. Similarly, given N_{cs} copysets, the probability of incurring any data loss (i.e., losing all the nodes in a copyset) is given by Equation 6.2.

$$P_{cs} = 1 - \left(1 - \frac{\binom{F}{R}}{\binom{N}{R}}\right)^{N_{cs}} \tag{6.2}$$

Two conclusions can be deducted from Equation 6.2. Firstly, the copyset-based scheme gives a smaller probability of data loss than the random replication, i.e., $P_{cs} < P_{ran}$, which is due to $N_{cs} < N_{su}$. Secondly, given the same R, N and F, then P_{cs} is an increasing function of N_{cs}. It means reducing the number of copysets can lead to a decrease in the probability of data loss. This conclusion is consistent with Figure 6.1.

Minimum Number of Copysets

In a copyset-based scheme, generating a minimum number of copysets can lead to a minimised probability of data loss. Note that each node should be assigned to at least $\frac{S}{R-1}$ copysets. Theoretically, for a cluster of N nodes, the minimum number of copysets is given by Equation 6.3, wherein $\frac{S}{R-1}N$ is divided by R because each copyset is counted R times repeatedly. Therefore, adding (or removing) a node requires at least $\frac{S}{(R-1)R}$ copysets to be created (or dismissed).

$$MinCopysets(N) = \frac{S}{R-1}\frac{N}{R} \tag{6.3}$$

The challenge of achieving the minimal copysets lies in ensuring that any two copysets overlap by at most one node. As defined in Equation 6.3, each node (e.g. n_i) is assigned to exactly $\frac{S}{R-1}$ copysets, which contain *at most* S distinct nodes other than n_i. Amongst these S nodes, if there exists a duplicate node, then n_i has only $(S-1)$ other distinct nodes to share its data. In this case, one more copyset is created for n_i to fulfil the requirement that it should share with S other distinct nodes. Then, the number of copysets will exceed the minimum number.

Moreover, while using a smaller scatter width S always leads to a lower probability of data loss, there are trade-offs to consider (Cidon et al., 2013). First, using a higher S reduces the average amount of data loss in a single failure event. That is, it trades off the frequency against the magnitude of data loss. Second, a higher S also means that a failed node can be recovered by more nodes. That is, this trade-off is between the loss frequency and the speed of recovery.

In this chapter, we focus on designing a copyset-based scheme that minimises the number of copysets. Moreover, the elasticity characteristic of IaaS Cloud means that nodes are dynamically added or removed from the system. The minimum number of copysets should be maintained under dynamic node changes.

6.3 Design of ElasticCopyset

This section describes the design of ElasticCopyset, a replication scheme that efficiently creates and dismisses copysets as the nodes are added or removed dynamically, such that the number of copysets is close to minimum (Equation 6.3) for higher data durability against multiple node failures.

6.3.1 The Intuition

There are properties that form the basis for an elastic, minimal copyset scheme. First, given R and S defined in Subsection 6.2.1, each copyset contains R nodes, and each node belongs to at least $\frac{S}{R-1}$ copysets, in which there are at least S other distinct nodes. Second, the addition or removal of a node affects only S other nodes in the $\frac{S}{R-1}$ copysets it belongs to. Last but more important, any two copysets overlap by at most one node, as the copyset-based scheme prefers more distinct nodes in fewer copysets.

To achieve these properties, we propose that the nodes should be split into groups, and that each node forms copysets only with nodes in the same group. There are advantages of isolating nodes into groups. First, as the number of nodes changes, the number of copysets can be changed by forming or dismissing a group of nodes accordingly. Second, the effect of adding or removing a node is restricted to one group, rather than the entire system.

As shown in Figure 6.3, a cluster of N nodes is divided into a list of groups, each having the same number of nodes, denoted as N_G. Thus, every N_G nodes form one *complete group*,

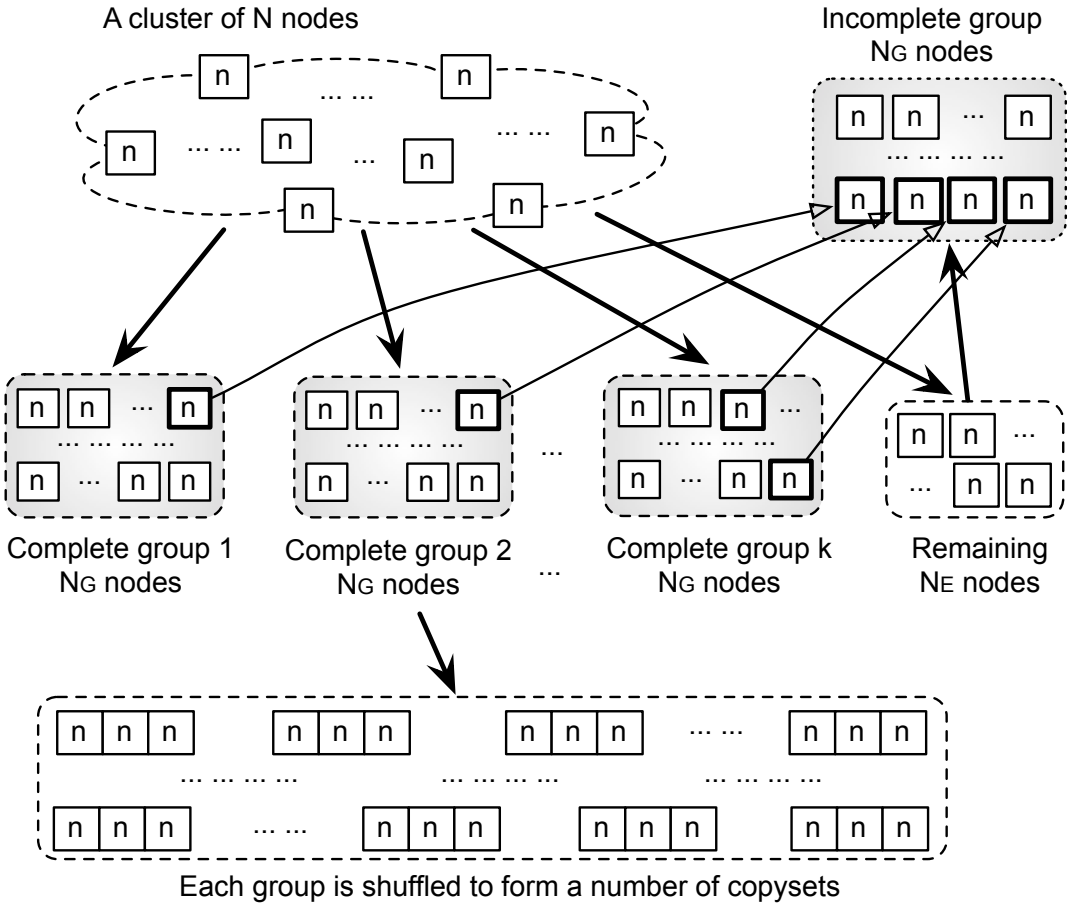


Figure 6.3: Dividing a cluster of nodes into complete and incomplete groups

wherein the minimum number of non-overlapping copysets are generated using a shuffle algorithm presented in Subsection 6.3.2.

However, the number of nodes N can be dynamic, and is usually not a multiple of N_G. There are extra nodes, $N_E = (N \mod N_G)$, wherein $0 \leq N_E < N_G$. The N_E nodes are insufficient to form a new complete group. When $N_E \neq 0$, ElasticCopyset randomly selects $(N_G - N_E)$ other nodes that are *already* in a complete group, to compensate the N_E nodes, so that one extra group of N_G nodes can be formed. This group is called the *incomplete group*, since there are nodes selected from other complete copyset groups. ElasticCopyset uses the same shuffle algorithm to generate copysets within the incomplete group.

Hence, if we group every N_G nodes for a cluster of N nodes, there are $\lfloor \frac{N}{N_G} \rfloor$ complete groups. One incomplete group is created, if there are extra nodes that cannot form a complete group, i.e. $N_E = (N \mod N_G) > 0$. Hence, there are $\lceil \frac{N}{N_G} \rceil$ groups in total. Note that $\lfloor \frac{N}{N_G} \rfloor$ and $\lceil \frac{N}{N_G} \rceil$ represent the floor and ceiling values of $\frac{N}{N_G}$, respectively.

In our shuffle algorithm, the value of N_G is defined using a theorem described in Subsection 6.3.3. Moreover, a replacement algorithm that handles node addition and removal in these groups is presented in Subsection 6.3.4.

6.3.2 Copyset Generation within a Group

ElasticCopyset generates copysets with a shuffle algorithm that uses three distinct shuffle orders. It requires that the replication number $R = 3$, which is the default value for many distributed data stores (Chang et al. 2008, Lakshman & Malik 2010, Ousterhout et al. 2010, Shvachko et al. 2010). Cidon et al. (2013) had evaluated the effect of varying replication number against the data durability and system performance, and reported that "(i.e. in random placement) increasing $R = 3$ to 4 does not provide sufficient durability, while using $R = 5$ or more significantly hurts the system's performance and almost doubles the cost of storage". Hence, we use $R = 3$.

Figure 6.4 illustrates the shuffle algorithm of generating copysets for one group of nodes. It consists of four steps: divide, replicate, shuffle and merge. First of all, the nodes are equally *divided* into $R = 3$ buckets, denoted as Bucket a, b, and c, respectively. Therefore, it requires that N_G is a multiplier of R. Second, each bucket is transformed into a matrix, wherein each node is *replicated* by $\frac{S}{R-1}$ times, because each node should appear in at least $\frac{S}{R-1}$ copysets. Third, the resulting matrices are *shuffled* in three different orders, namely Order 1, 2 and 3, respectively. Finally, given the location of a node in the matrix (i.e. the x^{th} row, the y^{th} column), each shuffled matrix offers one node located at the same position (x, y). These three nodes are *merged* to form one copyset.

Now that the intuition of the shuffle algorithm is presented, we provide the formal definitions for the parameters used, so that we can discuss how to achieve a minimum number of copysets. To begin with, the number of nodes in each bucket is denoted as C. Since each group is equally divided into $R = 3$ buckets, then $C = \frac{N_G}{R}$. The i^{th} node in each of the three buckets is denoted as $a[i]$, $b[i]$, and $c[i]$, respectively, where $0 \leq i < C$.

Next, the number of rows in the matrix is denoted as L, while the number of columns is equal to the node number in a bucket, i.e. C. As defined in Equation 6.4, $L = \frac{S}{R-1}$, because each node is replicated $\frac{S}{R-1}$ in the shuffle algorithm. In addition, the three matrices *after* shuffle are denoted as m_a, m_b, and m_c, respectively. The node located at the x^{th} row, y^{th}

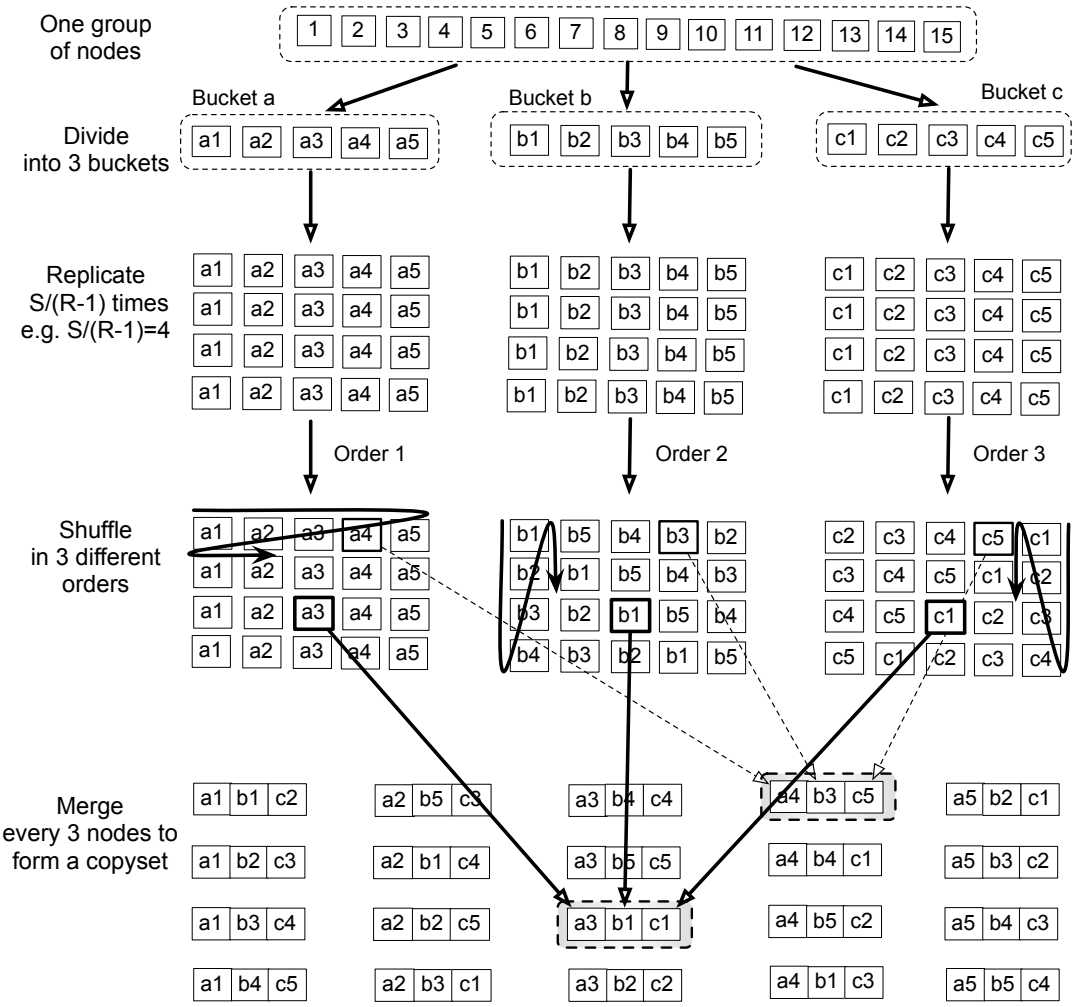


Figure 6.4: Create a minimum group of non-overlapping copysets by shuffling. In this example, there are $N_G = 15$ nodes, divided into $R = 3$ buckets: $\{a1, a2, a3, a4, a5\}$, $\{b1, b2, b3, b4, b5\}$ and $\{c1, c2, c3, c4, c5\}$. The scatter width $S = 8$, thus $\frac{S}{R-1} = 4$. Hence, the three buckets are replicated into three 4×5 matrices, which are shuffled to form non-overlapping copysets.

column is denoted as $m[x][y]$, wherein $0 \leq x < L$ and $0 \leq y < C$.

$$\begin{cases} C = \frac{N_G}{R} \\ L = \frac{S}{R-1} \end{cases} \tag{6.4}$$

Moreover, Figure 6.4 shows that the three matrices are shuffled in three different orders. In the following, we describe how to place the nodes from each bucket into the corresponding shuffled matrix. For each node, k is an integer that increases from 0 to $L - 1$, inclusively.

- **Order 1**. Left-to-right, then up-to-down. That is, the nodes are placed in order horizontally, in each and every row of the matrix. In mathematical terms, each $a[i]$

is placed at all positions of $m_a[x][y]$, where $\begin{cases} x = k \\ \\ y = i \end{cases}$.

- **Order 2**. Up-to-down, then left-to-right. That is, the nodes are place in order vertically, from the leftmost column to the rightmost column. In mathematical terms, each $b[i]$ is placed at all positions of $m_b[x][y]$, where $\begin{cases} x = (i + k * C) \mod L \\ \\ y = \lfloor \frac{i+k*C}{L} \rfloor \end{cases}$.
- **Order 3**. Up-to-down, then right-to-left. That is, the nodes are place in order vertically, from the rightmost column to the leftmost column. In mathematical terms, each $c[i]$ is placed at all positions of $m_c[x][y]$, where $\begin{cases} x = (i + k * C) \mod L \\ \\ y = C - 1 - \lfloor \frac{i+k*C}{L} \rfloor \end{cases}$.

Finally, each copyset is formed given by a pair of (x, y), wherein $0 \leq x < L$ and $0 \leq y < C$. That is, every three nodes $\{m_a[x][y], m_b[x][y], m_c[x][y]\}$ form a copyset. For example, as shown in Figure 6.4, given $(x = 2, y = 2)$, then $\{a3, b1, c1\}$ form a copyset, whilst given $(x = 0, y = 3)$, then another copyset $\{a4, b3, c5\}$ is generated. Note that the three matrices are of the same size. Hence, the number of copysets generated in one group is equal to the number of nodes in the matrix, i.e., $L * C$. The copyset number in a group, denoted as $CopysetsInGroup(N_G)$, is given by Equation 6.5 as a function of N_G.

$$CopysetsInGroup(N_G) = L * C = \frac{S}{R-1}\frac{N_G}{R} \tag{6.5}$$

By comparing Equation 6.5 with Equation 6.3, it can be deducted that this shuffle algorithm generates the minimum number of copysets for the group of N_G nodes. As discussed, for a cluster of N nodes, there are $\lceil \frac{N}{N_G} \rceil$ groups in total. Therefore, the total number of copysets generated by ElasticCopyset $TotalCopysets(N, N_G)$, is given by Equation 6.6 as a function of N and N_G.

$$TotalCopysets(N, N_G) = \frac{S}{R-1}\frac{N_G}{R}\lceil \frac{N}{N_G} \rceil \tag{6.6}$$

Compared to Equation 6.3, ElasticCopyset generates more copysets than the minimum. The difference (i.e. number of extra copysets) is given by Equation 6.7, which is linear with S and N_G.

$$ExtraCopysets(N, N_G) = \frac{S}{R-1}\frac{N_G - N_E}{R} \tag{6.7}$$

6.3.3 A Theorem for the Minimum Group

The key design of ElasticCopyset is to determine the minimum number of nodes required to form a group. As shown in Equation 6.7, using a smaller N_G reduces the number of extra copysets, thus producing close to the minimum number of copysets required, which reduces the probability of incurring data loss under simultaneous failures.

However, it is also constrained by the requirement that each node share copysets with S other distinct nodes. ElasticCopyset divides the nodes into a list of groups. In each group, each node is shuffled into $\frac{S}{R-1}$ copysets, in which there are exactly S nodes other than this node. Yet, it is uncertain that these S nodes are mutually distinct. Hence, our task is to find out the minimum value of N_G, such that there are exactly S other distinct nodes in the $\frac{S}{R-1}$ copysets to which each node belongs. Such N_G is given by Theorem 1.

Theorem 1. *Given the scatter width S, the replication number R, and a group of $N_G = R*C$ nodes. Let $L = \frac{S}{R-1}$. When C is the smallest odd number that is greater than L, the shuffle algorithm creates $\frac{S}{R-1}$ copysets for each given node, which is shared with exactly S other distinct nodes.*

In the following, we will prove Theorem 1 by deduction. The discussion is on choosing the appropriate value of C, such that any two nodes from each bucket (i.e. $a[i]$, $b[j]$, or $c[k]$, where $0 \leq i, j, k < C$), after shuffled in each corresponding order described on page 156, do not share more than one copyset. Note that every three nodes $\{m_a[x][y], m_b[x][y], m_c[x][y]\}$ form a copyset, given the same pair of (x, y), wherein $0 \leq x < L$ and $0 \leq y < C$.

Possible Values of C for Non-overlapping Copysets

As we think of the shuffle algorithm in column-wise, each node of Bucket a, after shuffled in Order 1, always and only appears in the i^{th} column of the resulting matrix m_a. In contrast, the column localities of the nodes from Bucket b or Bucket c depend on the value of C, which is constrained by Lemma 1. The formal proof is presented in Appendix A.1.

Lemma 1. *When $C \geq L$, given $0 \leq i < C$, any $b[i]$, after shuffled by L times in Order 2 (or $c[i]$ shuffled in Order 3), will appear at most once in any column of the shuffled matrix.*

Lemma 1 points out that, when $C \geq L$, any two replicas of $b[i]$ (or $c[i]$), will not be allocated in the same column of its corresponding shuffled matrix (i.e. m_b or m_c). However, all the replicas of $a[i]$ always appear in the same column. Since any two nodes

from different columns will not be *merged* into the same copyset, Lemma 2 can be deducted from Lemma 1, and has been proved in Appendix A.2.

Lemma 2. *When $C \geq L$, given $0 \leq i, j, k < C$, any combination of $(a[i], b[j])$ or $(a[i], c[k])$ appears in at most one copyset.*

Moreover, as we consider the shuffle algorithm in row-wise, each $a[i]$ appears in each and every row of m_a, while the row localities of $b[i]$ and $c[i]$ still depend on C. More specifically, the row position $x = (i + k * C) \mod L$. Similarly, if each $b[i]$ (or $c[i]$) is also placed in each and every row of the shuffled matrix m_b (or m_c), it will simplify the operation of restricting any two nodes from being allocated together more than once. According to Lemma 3, C and L should be co-prime (i.e. mutually prime). The proof is given in Appendix A.3.

Lemma 3. *When $C \geq L$, if C and L are co-prime, each node (either $a[i]$, $b[i]$, or $c[i]$), after shuffled by L times (either in Order 1, 2 or 3), will appear once in each and every row of the shuffled matrix.*

Lemma 3 is an important step towards restricting any combination of $(b[i], c[j])$ (where $0 \leq i, j < C$) from being allocated into multiple copysets. The reason is that, when each replica of $b[i]$ or $c[j]$ is restricted to one specific row, the problem can be simplified as preventing $(b[i], c[j])$ from appearing on the same column more than once, which is discussed as follows.

The Minimum C for Non-overlapping Copysets

According to Lemma 3, it requires that $C \geq L$, and C is prime to L. Thus, the minimum value of C is the smallest odd number that is greater than L. We have proved that, given such value of C, any combination of $(b[i], c[j])$ appears in at most one copyset, as depicted in Lemma 4. Note that the value of L is determined by S and R, i.e. $L = \frac{S}{R-1}$. The proof presented in Appendix A.4 is consisting of two parts based on two possible values of L: i) an even integer; ii) an odd integer. Under both circumstances, there does not exist a combination of $(b[i], c[j])$ that appears in more than one copyset.

Lemma 4. *When C is the smallest odd number greater than L, given $0 \leq i, j < C$, any combination of $(b[i], c[j])$ appears in at most one copyset.*

Hence, based on Lemma 2 and 4, given $0 \leq i, j, k < C$, when the value of C is the smallest odd number that is greater than L, then any combination of $(a[i], b[j])$, or $(a[i], c[k])$, or $(b[j], c[k])$, appears in at most one copyset, as depicted in Lemma 5, the proof of which is presented in Appendix A.5). Given such C, each node shares copysets with exactly S other distinct nodes in each group, as depicted in Lemma 6, proved in Appendix A.6.

Lemma 5. *When C is the smallest odd number greater than L, any two copysets share at most one common node.*

Lemma 6. *When C is the smallest odd number greater than L, every node belongs to exactly L copysets, in which there are exactly S other distinct nodes.*

Till this end, we have proved Theorem 1 in mathematical terms. It demonstrates that a collection of minimal copysets can be created for a relatively small group of nodes. Hence, according to Theorem 1, the minimum number of nodes in each group is $N_G = R * C$, wherein $C = \frac{S}{R-1} + 1$ if $\frac{S}{R-1}$ is even; otherwise $C = \frac{S}{R-1} + 2$. Using a minimised group can ensure a close to minimised probability of data loss.

6.3.4 Handling Node Changes

In a distributed storage system running on an IaaS Cloud, nodes can spontaneously join or leave the system. ElasticCopyset aims at minimal disruption of existing copysets when handling node addition and removal.

As depicted in Figure 6.3 (page 154), the incomplete group is consisting of the remaining N_E nodes, plus $(N_G - N_E)$ nodes randomly selected from the complete groups, together making a total of N_G distinct nodes. Hence, according to the type of group to which a node belongs, we have defined three node roles, described as follows.

- **Extra**: the node is in the incomplete group, and not in any complete group.
- **Elementary**: the node is in a complete group, and not in the incomplete group.
- **Supplementary**: the node is in both complete and incomplete groups.

According to this definition, an incomplete group is comprised by both *Extra* and *Supplementary* nodes. In contrast, a complete group consists of *Elementary* nodes, but it may or may not have *Supplementary* nodes, which are randomly chosen by the incomplete group. Moreover, since each *Supplementary* node is placed into two groups, it is allocated

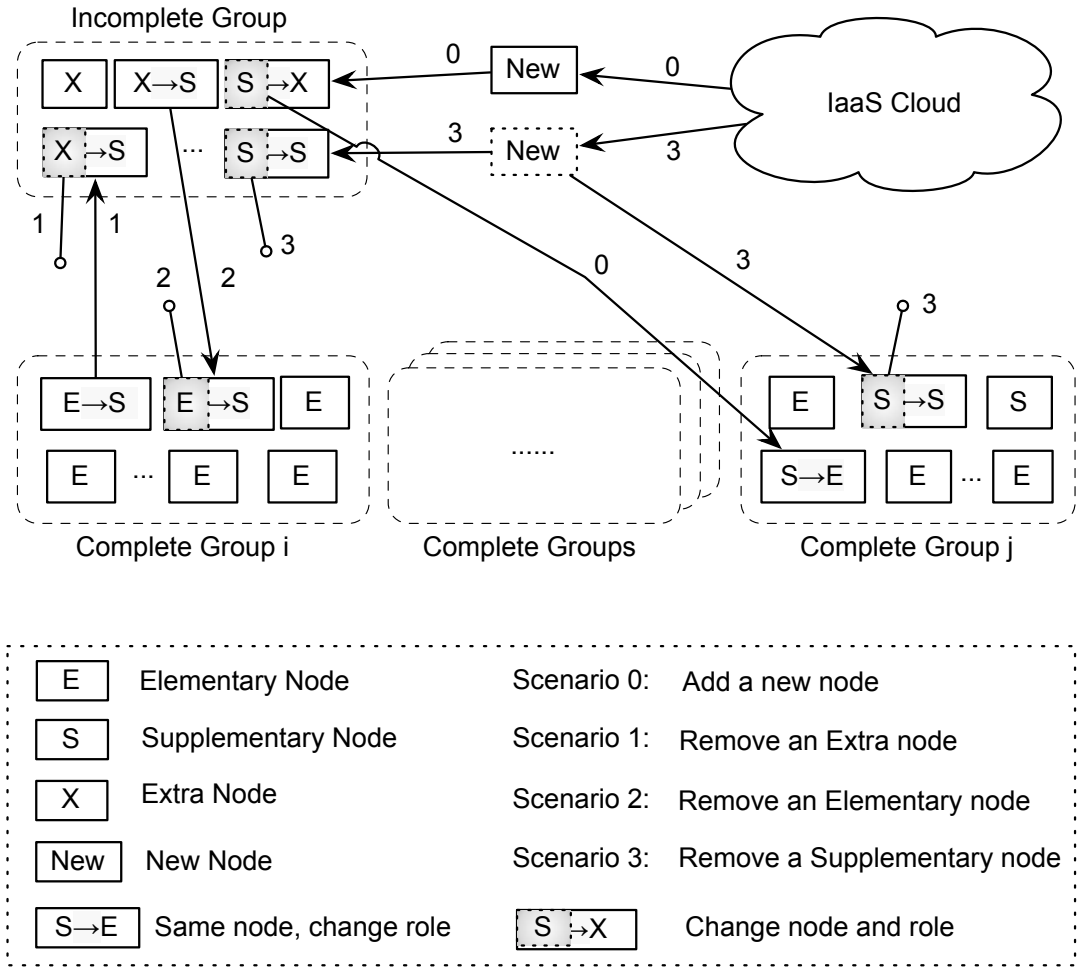


Figure 6.5: Adding and removing a node in ElasticCopyset

to $\frac{2S}{R-1}$ copysets, rather than $\frac{S}{R-1}$. The total number of extra copysets introduced by these *Supplementary* nodes has been given by Equation 6.7 on page 157. This number is minimised by choosing a minimum group N_G defined by Theorem 1.

Node Addition and Removal in Groups

Figure 6.5 depicts the operations to deal with node changes. The change of nodes is handled according to the role of the node. Since there are three node roles, node removal (either consciously or unexpectedly), is dealt with under three scenarios, labeled as 1, 2, and 3, respectively. In contrast, there is only one scenario for node addition, labeled as 0.

- **Scenario 0** illustrates the addition of a new node. As shown, the new node always joins the incomplete group. Since there are already N_G nodes, one of the *Supplementary* nodes is chosen and replaced by the new node. According to the definition, the chosen *Supplementary* node is also located in a complete group. Hence, this

node changes its role to *Elementary* in the complete group. By comparison, the new node is marked as *Extra* in the incomplete group.

- **Scenario 1** illustrates the removal of an *Extra* node from the incomplete group. In this case, an *Elementary* node is randomly selected from one complete group to replace the removed node. The selected node is marked as *Supplementary* in both the complete and incomplete groups.

- **Scenario 2** illustrates the removal of an *Elementary* node from a complete group. In this case, the incomplete group offers one of its *Extra* nodes. This node replaces the removed node in the affected complete group. Now that the node is assigned to both the complete and incomplete groups, it is marked as *Supplementary* in both groups.

- **Scenario 3** illustrates the removal of an *Supplementary* node, which is located in the incomplete group and one complete group. There are two solutions. The first solution consists of two steps: i) select an *Elementary* node to replace the removed node in the incomplete group as in Scenario 1; ii) select an *Extra* node to replace the removed same node in the complete group as in Scenario 2. This approach is involved with two operations of node replacement. An alternative is to provision a new node from the IaaS Cloud, and use it to replace the removed node in both groups, which is depicted in Figure 6.5.

Moreover, there are extreme cases that involve with the creation and dismissal of the incomplete group. As depicted in Figure 6.6, due to consecutive node additions, there may be no more *Supplementary* node left for replacement. That is, every node in the incomplete group is *Extra*. In this case, the existing incomplete group is marked as *complete*, with all the nodes in it marked as *Elementary*. In the meanwhile, a new incomplete group is created, and the new node joins it as an *Extra* node. To populate the new incomplete group, $(N_G - 1)$ *Elementary* nodes are randomly selected from all the complete groups, such that there are N_G nodes in total. These selected nodes are marked as *Supplementary*.

Conversely, during node removal as in **Scenario 2**, if the incomplete group has no more *Extra* node to offer, then it is dismissed. As depicted in Figure 6.7, the complete group in which the removed node is located, becomes the new incomplete group, wherein all the existing nodes become the *Extra* nodes. Therefore, there are $(N_G - 1)$ *Extra* nodes.

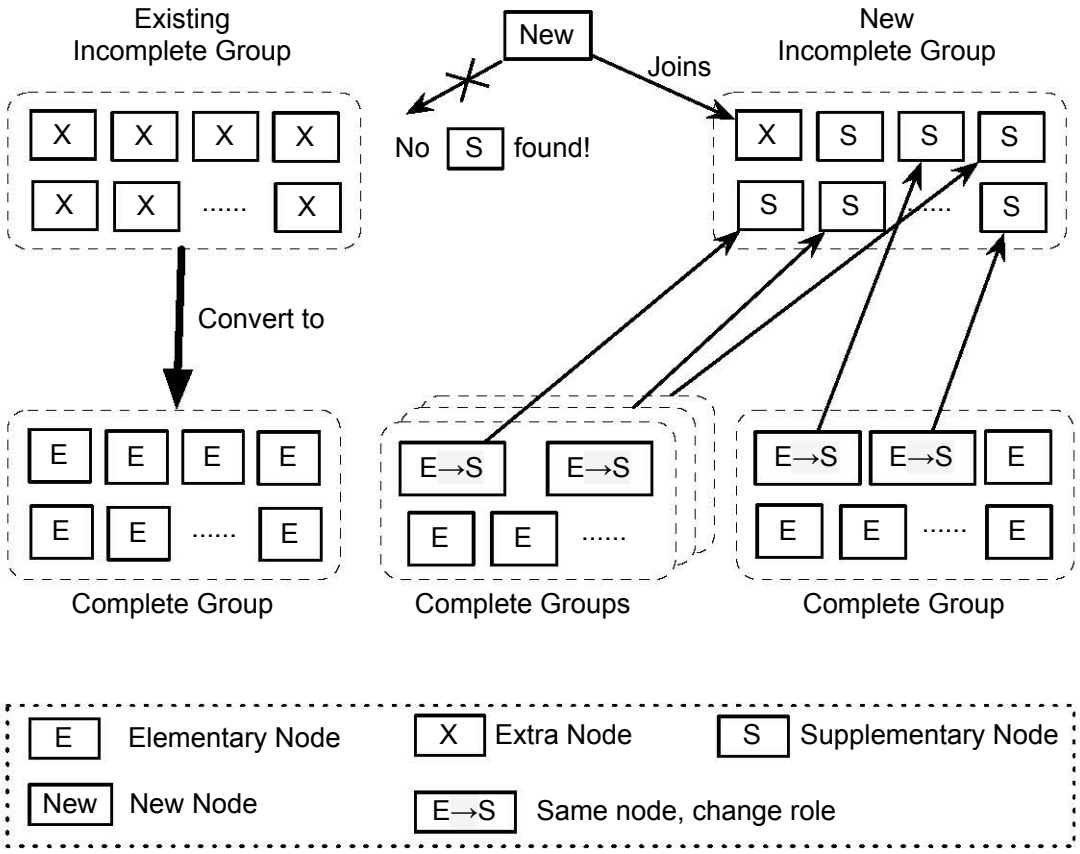


Figure 6.6: Create a new incomplete group in ElasticCopyset

One more *Elementary* node is randomly selected from a complete group, and is used to replace the removed node. Hence, this selected node is marked as *Supplementary*.

Node Changes in Copysets

In the previous discussion, there are three operations that require the change of copysets: i) replacing a node in one group; ii) creating a new incomplete group; and iii) dismissing an existing incomplete group. Note that changing the role of a node does not affect any copyset. In the following, we discuss the side-effects of these operations to the stability of copysets.

First of all, node replacement is inevitable in every node addition and removal described in Figure 6.5. In most cases, ElasticCopyset requires only one node replacement, as in Scenario 0, 1, and 2. Even in Scenario 3 where a *Supplementary* node is removed, it requires only two node replacements. Note that replacing a node in one group affects $\frac{S}{R-1}$ copysets that the node is associated with. To execute a node replacement, the data belonging to the replaced node, is transferred to the newly joined node from the existing

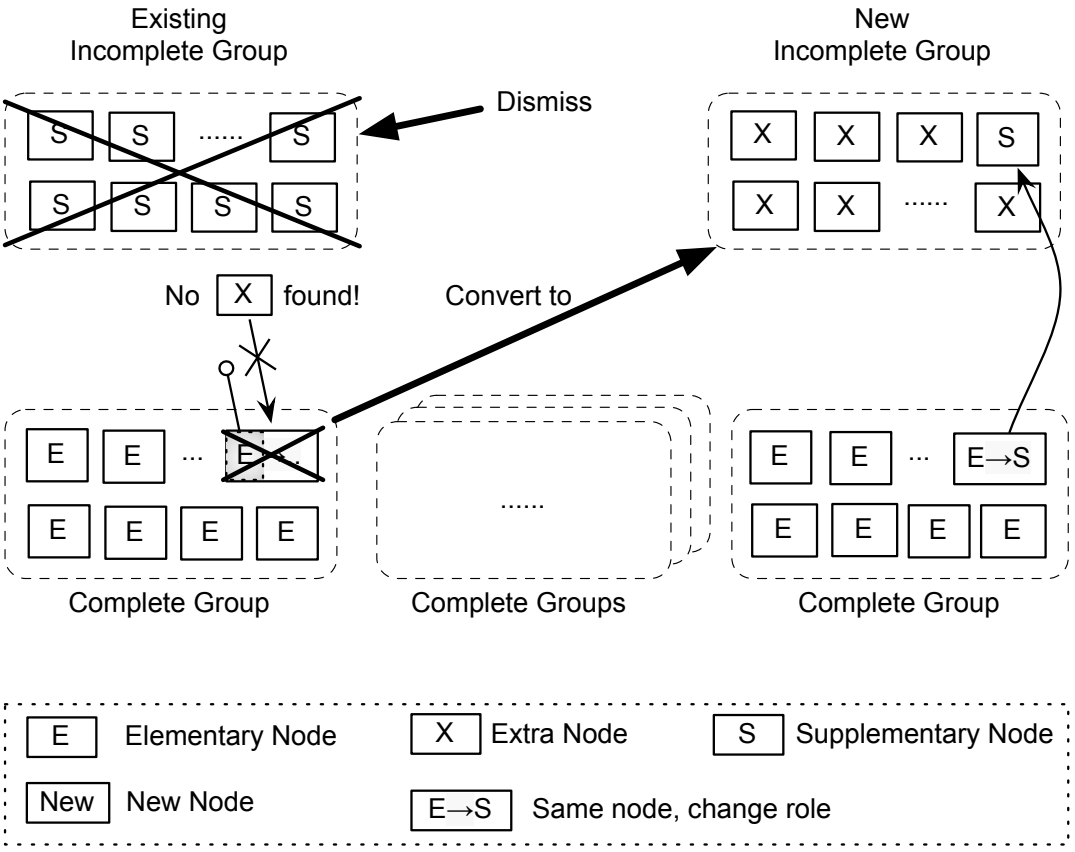


Figure 6.7: Dismiss an existing incomplete group in ElasticCopyset

S nodes in the affected $\frac{S}{R-1}$ copysets.

Moreover, creating a new incomplete group (as in Figure 6.6) requires the generation of $\frac{S}{R-1}\frac{N_G}{R}$ copysets (Equation 6.5, page 157). In this case, although there are $\frac{S}{R-1}\frac{N_G}{R}$ copysets involved, data movement is only required in the $\frac{S}{R-1}$ copysets that contain the new node. That is, it affects only S existing nodes.

Conversely, unlike creating a group, dismissing an existing incomplete group (as in Figure 6.7) does involve data movement for all its $\frac{S}{R-1}\frac{N_G}{R}$ copysets. Nevertheless, the incomplete group is dismissed because all its nodes have become *Supplementary*. In order to reduce the amount of data moved during this process, ElasticCopyset requires that no data should be assigned to the copysets that are in the incomplete group whilst having no *Extra* node. Hence, the data is moved out gradually as more *Extra* nodes become *Supplementary*. By the time when the incomplete group is to be dismissed, there is already no data left.

To sum up, in ElasticCopyset, each node addition or removal requires data movement in $\frac{S}{R-1}$ copysets, which involves only one scatter width of nodes. In the extreme cases

when both the incomplete and complete groups are involved, it affects $\frac{2S}{R-1}$ copysets that contain $2S$ nodes other than the added/removed node.

6.3.5 Discussion on Implementing ElasticCopyset

ElasticCopyset is a general-purpose data placement scheme that can be implemented on a wide range of distributed storage systems that distribute data across a cluster of nodes. This subsection discusses its implementation on decentralised shared-nothing KVSs as described in this thesis.

First and foremost, ElasticCopyset is designed for large-scale distributed systems with dynamic node arrivals and departures. It requires at least one complete group to create minimal copysets, with one incomplete group to deal with node changes. Therefore, the scatter width should be properly set, so that the existing nodes in the system are sufficient to form at least one complete group of nodes, i.e., $N \geq N_G$.

According to Theorem 1 (page 158), the number of nodes in one group is $N_G = R * C$, wherein C equals $(\frac{S}{R-1} + 1)$ or $(\frac{S}{R-1} + 2)$, such that C is an odd number. Hence, given $R = 3$, the size of N_G is determined by S. If the scatter width S is set too high, such that the total number of nodes $N < N_G \leq R(\frac{S}{R-1} + 2)$, then even one minimum group cannot be formed. To avoid this situation, ElasticCopyset requires that $S \leq Min(N)/2$, wherein $Min(N)$ is the minimum number of nodes that will be maintained in the system. Such requirement does not hurt the system's performance, because S is usually set to a lower value to reduce the probability of data loss.

Initialisation of Copysets

The group size $N_G = R * C$ is calculated based on S that is set. As depicted in Figure 6.3 (page 154), every N_G nodes form one complete group, while the nodes that remain in the end are assigned to the incomplete group. However, there is one implementation challenge, that is, the lack of centralised components to divide the nodes into groups at system startup.

To address this challenge, ElasticCopyset relies on the system administrators to designate at least one *group coordinator* at system initialisation. Each coordinator is responsible for populating one complete group of nodes, and therefore, the number of coordinators designated can be any positive number no greater than $\lfloor \frac{N}{N_G} \rfloor$, i.e., the number of complete

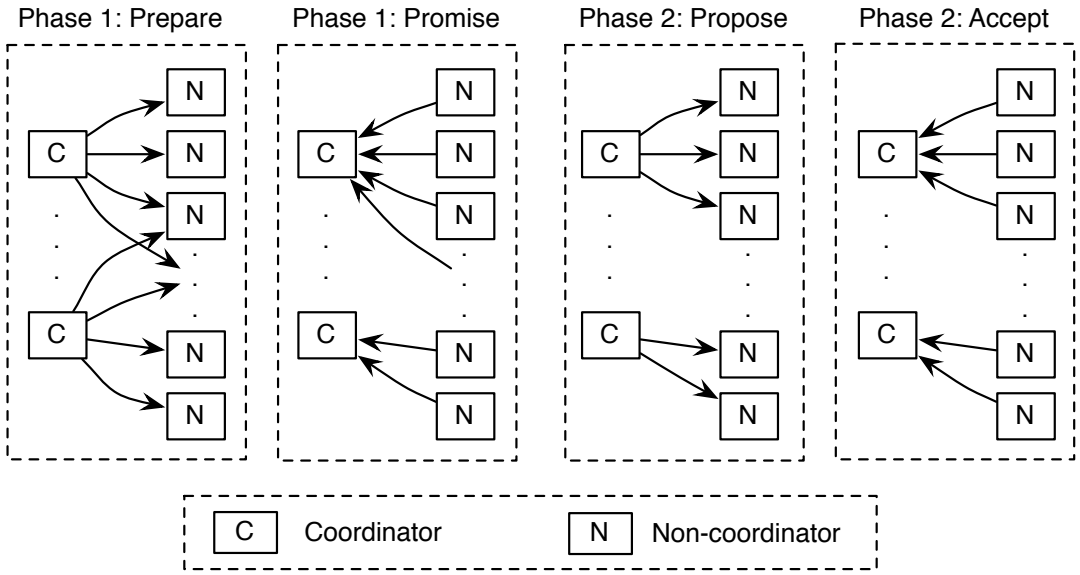


Figure 6.8: Populating the complete groups using Paxos-based coordination

groups.

To populate a complete group, each coordinator uses the typical two-phase coordination in Paxos (Lamport, 2001): i) prepare/promise, and ii) propose/accept, as depicted in Figure 6.8. In Phase 1, each coordinator broadcasts a message to all the non-coordinator nodes, requesting them to *prepare* for joining its group. Each non-coordinator node can receive multiple *prepare* requests from different coordinators, but it can *promise* to only one coordinator that it will not accept any proposal from other nodes within a time period (i.e. the release). Thus, at the end of Phase 1, each coordinator has collected a number of promises from the non-coordinators. Next, in Phase 2, each coordinator chooses at most $(N_G - 1)$ nodes from the promises, and *propose*s to these chosen nodes for joining its group. On receiving the message, the non-coordinator node will *accept* the proposal if its previous promise is still valid.

In this way, each coordinator enrols a number of nodes into its group. However, in Phase 1, some coordinators may have collected fewer promises than $(N_G - 1)$, and as a result, cannot gather enough nodes to form a complete group. To resolve this issue, if a coordinator has already enrolled $(N_G - 1)$ non-coordinator nodes, it sends out another message to decline the promises that are not chosen by itself, so that those non-coordinator nodes become available for other coordinators that need more nodes. Therefore, a coordinator can repeatedly run the process of enrolling nodes, until it has acquired $(N_G - 1)$ nodes.

Note that when the number of coordinators designated is less than $\lfloor \frac{N}{N_G} \rfloor$, there must be at least N_G non-coordinator nodes that are not assigned to any group. To create more complete groups, the existing coordinators will elect one more coordinator amongst the remaining non-coordinators, wherein the election can be the same as the one proposed in Subsection 4.2.2. Thus, the elected coordinator follows the two-phase coordination (Figure 6.8), and forms one more complete group of N_G nodes. At this moment, if the number of remaining non-coordinator nodes is still no less than N_G, then another new coordinator is elected by the existing coordinators. This process can be repeatedly executed, until there are less than N_G nodes remaining. Finally, with the help of $\lfloor \frac{N}{N_G} \rfloor$ coordinators that are designated or elected, $\lfloor \frac{N}{N_G} \rfloor$ complete groups are formed.

Note that there are still $N_E = (N \mod N_G)$ nodes that are not chosen. Once all the coordinators have finishing populating its own complete group, they also coordinate to form an incomplete group for the remaining nodes. Similarly, one new coordinator is elected amongst the remaining nodes. The elected coordinator enrols the remaining nodes automatically, and it also randomly selects more nodes from the complete groups to join the incomplete group as *Supplementary* nodes. In the end, one incomplete group of N_G nodes is also formed.

Once the nodes have been divided into groups, each group coordinator sorts its group members based on certain criteria. Then, each group member is assigned with a ranking position in the group, and uses this position p to calculate the copysets it belongs to. For example, as in Figure 6.4 (page 156), if a node is ranked as $p = 7$ out of the 15 nodes, it immediately knows that it is placed at the second position of the second bucket, i.e. $b2$ in Figure 6.4. Next, it runs the shuffle algorithm, which shows that it belongs to four copysets: $\{a1, b2, c3\}$, $\{a2, b2, c5\}$, $\{a3, b2, c2\}$, and $\{a5, b2, c1\}$.

In ElasticCopyset, each node is mapped to a group ID and a ranking position once it joins a group. This position remains unchanged until the node is removed from the group. Thus, given the information of node positions, every node can calculate the members of each copyset locally, without the presence of a centralised component. In addition, it is worth mentioning that, when a node replaces an existing node in a group, it simply takes over the ranking position of the node to be replaced, since the shuffle algorithm relies on the ranking positions to determine the copysets.

Replica Placement

In Subsection 4.2.3, a set of replica placement algorithms have been proposed to assign partition replicas to each individual node. In contrast, ElasticCopyset requires each storage unit (i.e. a partition replica) to be associated with one copyset of nodes. Therefore, these algorithms cannot be leveraged by ElasticCopyset directly, but the proposed rules (page 81) can be reused.

Note that each bootstrapping (or decommissioning) node is responsible for deciding the replicas that it will serve. In ElasticCopyset, each new node is added to the incomplete group, wherein it takes over the position of a *Supplementary* node as illustrated in Figure 6.5. Therefore, a bootstrapping node in ElasticCopyset will serve all the replicas that are already assigned to the copysets that it belongs to. In the meanwhile, it also attempts to balance the workload and distribution of data across the copysets. Similar to Rule 3 and 4 described on page 81, the bootstrapping node calculates the average CPU utilisation and data volume in each copyset, and then initiates the reallocation of replicas to/from its own copysets.

Apart from node bootstrapping and decommissioning, ElasticCopyset also deals with the creation and dismissal of the incomplete group. As discussed on page 164, although a new incomplete group creates $\frac{S}{R-1}\frac{N_G}{R}$ copysets, there are only $\frac{S}{R-1}$ copysets (that contain the new node) requiring data movement. Hence, the overhead of group creation is the same as node bootstrapping.

In contrast, to dismiss an incomplete group, all the $\frac{S}{R-1}\frac{N_G}{R}$ copysets in it are destroyed. To reduce the data volume transferred at group dismissal, we propose that, in the incomplete group, the copysets that have no *Extra* node should not be assigned with any data. To fulfil this requirement, once the only *Extra* node in a copyset is removed, the replicas assigned to this copyset should be moved out gradually (i.e. not affecting online queries) by the remaining *Supplementary* nodes in this copyset.

Recovery from Multiple Node Failures

ElasticCopyset is designed to maintain high data durability under simultaneous node failures. However, when there are F nodes that have failed, even if there is no data loss, the failure event can affect at most $\frac{S}{R-1}F$ copysets.

One solution is to provision F new empty nodes from the IaaS Cloud. These new nodes

are added one after another, each choosing a failed node to be replaced. Alternatively, if there is no new node available, we propose to reorganise the location of nodes across the groups. The challenge lies in minimising the disruption to the ranking positions of the living nodes in each group, because the shuffle algorithm relies on these node positions to deterministically map the nodes into copysets.

To reorganise the group location of the nodes when F nodes have failed, ElasticCopyset always uses the living *Extra* nodes in the incomplete group, to replace the failed nodes in the complete groups. To begin with, ElasticCopyset requires a coordinator elected amongst the nodes in the incomplete group. This coordinator calculates the number of *Extra* nodes that are still alive, denoted as N_L, and the number of failed nodes located in the complete groups, denoted as F_C. There are two scenarios depending on the values of these two numbers.

On one hand, if $N_L > F_C$, then there are enough living *Extra* nodes to fill the vacancies in the complete groups. To do so, the coordinator reallocates the *Extra* nodes one by one. Each *Extra* node is assigned to one complete group to replace the position of one failed node. Note that after an *Extra* node is settled down in a complete group, it is still present in the incomplete group, but is marked as *Supplementary* in both groups instead. In addition, it is worth mentioning that, this process does not require re-assignment between partition replicas and copysets, but does require the added node to take over the data belonging to the failed node. When all the F_C nodes are replaced, all the complete groups become integrated. In addition, if the number of living nodes in the incomplete group is less than N_G, then the coordinator will randomly select the *Elementary* nodes one by one from the complete groups, to fill the vacancies in the incomplete group.

On the other hand, if $N_L \leq F_C$, it means that the whole incomplete group is to be dismissed. To do so, the coordinator moves out the *Extra* nodes one by one. From each of the $\frac{S}{R-1}$ original copysets, the moving node carries $1/R$ of the replicas to each of its $\frac{S}{R-1}$ destination copysets located in the complete group. Note that this node is then marked as *Supplementary* in both the incomplete and complete groups. In this way, the data in the incomplete group is gradually moved out by each *Extra* node. Finally, all the *Extra* nodes have been moved out, and all the nodes in the incomplete group are *Supplementary*. Thus, this group is dismissed after the data is reallocated by the nodes that remain.

Next, if $F_C - N_L > 0$, it means that there is at least one complete group with vacancies

caused by failed nodes. The previous coordinator (from the incomplete group) chooses the complete group with the most vacancies, and converts it to be an incomplete group. This process is the same as shown in Figure 6.7 on page 164. A new coordinator is elected from this new incomplete group. Therefore, this new coordinator follows the same procedure as described, and uses its *Extra* nodes to compensate other complete groups. This procedure can go on iteratively, until all the vacancies are filled in the groups.

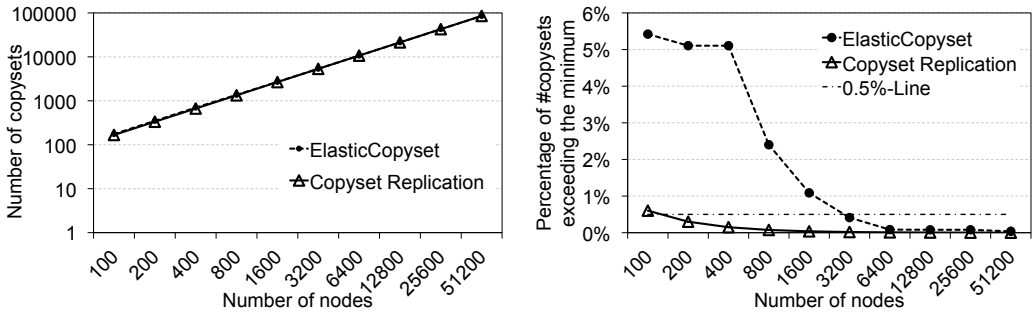
To sum up, in ElasticCopyset, each failed node is replaced by one *Extra* (or new empty) node, and the data volume transferred in each node replacement is equal to data volume in the failed node. Moreover, for each incomplete group that is dismissed, it requires at most $\frac{S}{R-1}\frac{N_G}{R}$ re-mappings between partition replicas and copysets.

6.4 Evaluation

In this section, we provide a set of experimental results to evaluate the impact of ElasticCopyset against Copyset Replication (Cidon et al., 2013) and Random Replication, on data durability under simultaneous node failures. We have used simulations to evaluate the algorithms, as we do not have access to the thousands of compute nodes needed for real-world experiments.

For all the experiments, the replication number is set as $R = 3$, as explained in Subsection 6.3.2. The number of nodes N changes between 1000 and 10000, and there are 10000 storage unit (e.g., data chunk, partition replica) assigned to each node. This is the typical cluster size at Facebook (Borthakur et al., 2011), LinkedIn (Chansler, 2012) and RAMCloud (Ongaro et al., 2011), and is also consistent with the experiments conducted in Copyset Replication (Cidon et al., 2013).

We have evaluated data durability under three scenarios: i) static deployment with varied settings; ii) linear scaling at a constant rate; and iii) elastic scaling based on workload demands. We have used the probability of data loss to quantify data durability. Each loss probability is calculated as follows. We simulate 10000 times of simultaneous node failures, given the same percentage of node failures. Each time, the given percentage of nodes are randomly chosen to fail. This failure event is considered to have caused data loss if there exists at least one copyset in which all nodes have failed. The probability of data loss is equal to the number of failure events that have caused data loss, divided by 10000.



(a) Number of copysets (b) Percentage exceeding the minimum copyset

Figure 6.9: Number of copysets generated by different placement strategies

6.4.1 Evaluation on Static Deployment

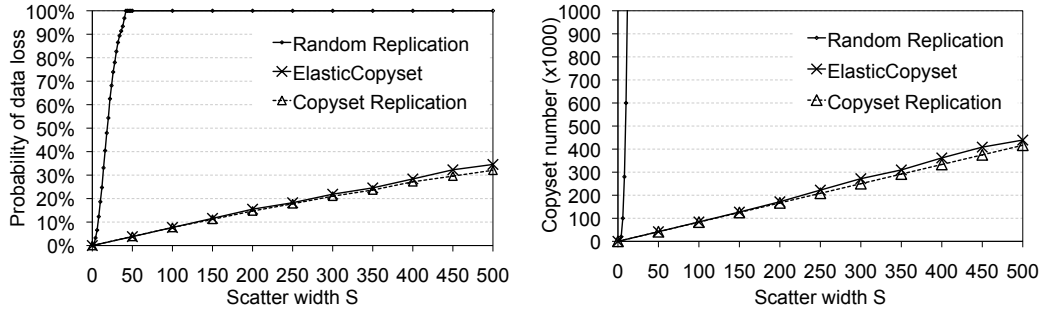
This set of experiments compare ElasticCopyset with the other placement schemes, using static deployment with varied settings of number of nodes, scatter width and percentage of nodes that failed.

Varied Numbers of Nodes

Figure 6.1 on page 148 has demonstrated that, the probability of data loss is dominated by the number of copysets in the system. In this experiment, we used different numbers of nodes, to study the number of copysets generated using different placement schemes. The number of nodes is doubled at every step, starting from 100 up to 51200. The scatter width is set as $S = 10$, which is consistent with that reported for Facebook deployment (Borthakur et al., 2011) and in the evaluation of Copyset Replication (Cidon et al., 2013).

Figure 6.9 compares the number of copysets generated by Copyset Replication and ElasticCopyset. Figure 6.9a shows that, the numbers of copysets generated in both schemes increase linearly with the number of nodes. This result is consistent with Equation 6.3 and 6.6, both equations showing that the number of copysets is a linear function of the node number.

In order to clearly differentiate the two numbers, we present Figure 6.9b, which compares the resulting number of copysets towards the minimum copysets defined in Equation 6.3. As can be seen, when the number of nodes $N \geq 100$, the percentage of copysets exceeding the minimum number is less than 0.5% using Copyset Replication. In contrast, ElasticCopyset generates slightly more copysets. When $100 \leq N \leq 400$, the number of



(a) Data loss probability, with 1% of nodes failed (b) Number of copysets

Figure 6.10: Testing how the varied scatter width S will impact the system of 5000 nodes

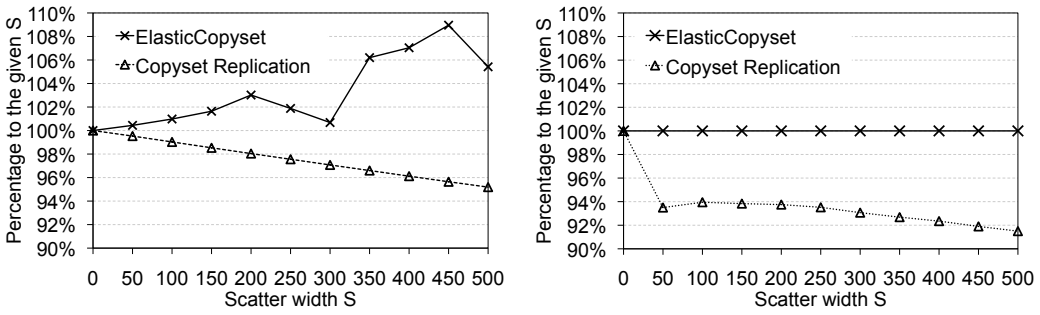
copysets generated exceeds the minimum by 5%. This percentage of excess falls below 0.5% when $N \geq 3200$. Overall, both schemes generate a close to minimum number of copysets.

Varied Scatter Widths

In a data placement scheme, the value of scatter width is determined by the system administrators. Figure 6.10 depicts how a system of 5000 nodes will be affected when the scatter width S increases from 0 up to 500.

Figure 6.10a depicts the probability of data loss when 1% of the nodes, i.e. 50 nodes, fail simultaneously. As can be seen, the probability of data loss using Copyset Replication and ElasticCopyset is below 40% even when $S = 500$, while with Random Replication, the data is almost guaranteed to be lost when $S > 50$. Figure 6.10b shows that the number of copysets generated by Copyset Replication and ElasticCopyset grows steadily and linearly as S increases. In contrast, the number of copysets generated by Random Replication goes beyond one million even when S is small, because Random Replication forms one copyset for each storage unit, and there are 10000 storage units per node. By comparing Figure 6.10a with Figure 6.10b, it shows that the probability of data loss is determined by the number of copysets in the system. This is also consistent with the conclusion of Equation 6.2 on page 152.

Moreover, Figure 6.11 compares the actual scatter width, i.e. number of distinct nodes that each node shares copysets with, against the value of S predefined by system administrators. Figure 6.11a shows the percentage of the average actual scatter width of all the nodes towards the defined S. As S increases, ElasticCopyset presents a wave-like ascent in the average scatter width, while Copyset Replication shows a steady decline in the actual



(a) Average scatter width

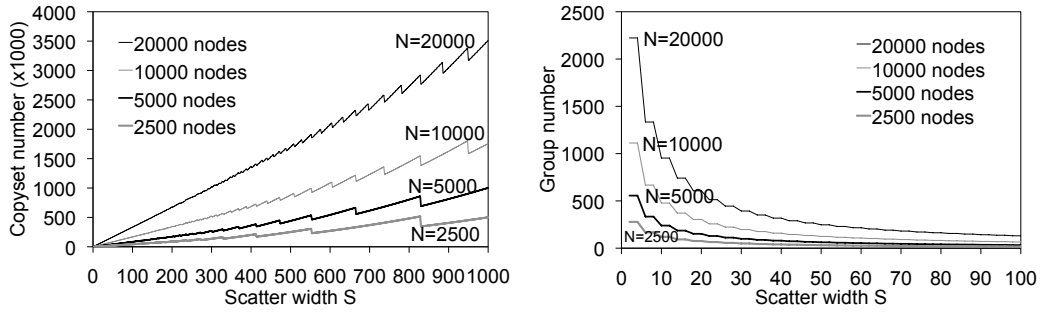
(b) Minimum scatter width

Figure 6.11: Compare the actual scatter width against the predefined S

scatter width on average. When $S = 500$, the average scatter width in ElasticCopyset is 105% of the defined S, comparing to 95% of S in Copyset Replication. Moreover, Figure 6.11b presents the minimum scatter width of all the nodes. ElasticCopyset presents a constant minimum scatter width equal to the given S, while Copyset Replication again shows a steady decline in the minimum scatter width. The latter is less than 94% of S when $S > 50$.

Overall, ElasticCopyset exhibits higher scatter width than Copyset Replication, but the percentages of deviation to the given S are less than 10% in both schemes. The result of Figure 6.11 can be explained. ElasticCopyset guarantees that each node is put into copysets with S distinct nodes, so the minimum scatter width is always equal to S. Moreover, the incomplete group generates extra copysets, the number of which is determined by both S and N_E (i.e. Equation 6.7), causing a non-linear increase in the average scatter width of ElasticCopyset. In contrast, Copyset Replication uses random permutation to form $\frac{S}{R-1}$ copysets for each node, without guarantee that there are S distinct nodes. A greater S results in more duplicating nodes in the copysets that a node belongs to. Hence, the average and minimum scatter widths both decrease as the given S grows.

In addition, Figure 6.12 studies ElasticCopyset on the changes in number of copysets and groups. In this experiment, the scatter width is increased from 0 up to 1000. There are four tests, with the number of nodes N set as 2500, 5000, 10000, and 20000, respectively. As shown, a larger scale of system results in more copysets and groups. Moreover, Figure 6.12a shows that the number of copysets rises in a sawtooth pattern as the scatter width S grows. By comparison, Figure 6.12b shows the number of groups for $S \leq 100$. As S increases from 0 to 100, the total number of groups drops quickly like a power law



(a) Number of copysets (b) Number of groups

Figure 6.12: The change in numbers of copysets and groups, given varying scatter widths and node numbers in ElasticCopyset

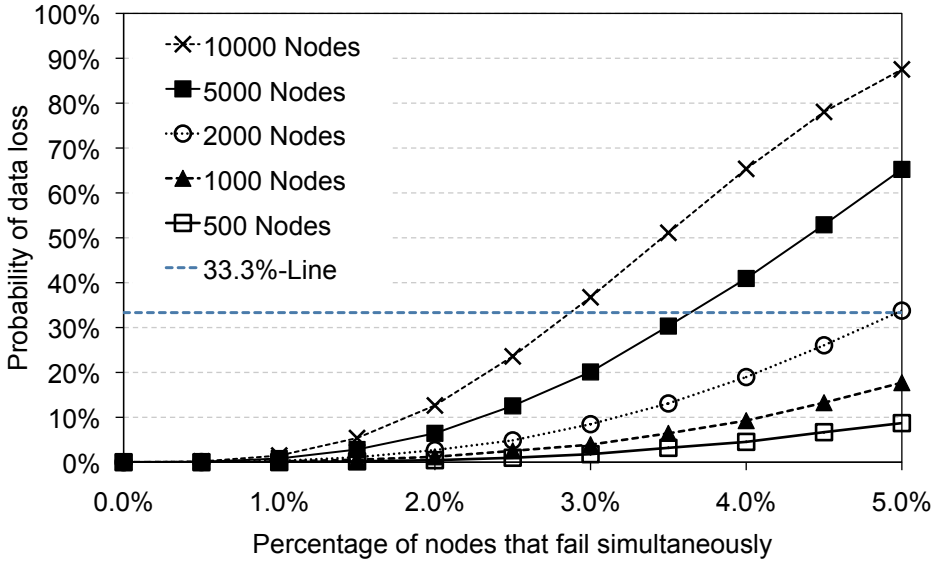


Figure 6.13: Probability of data loss with varying percentage of the nodes failed simultaneously, using different replication strategies, given S=10.

curve. We also observed that the group number declines slowly as a long tail from $S \geq 100$ onwards. In addition, it is worth mentioning that, the sawtooth pattern in both subfigures is due to the existence of an incomplete group, which creates extra copysets for the N_E nodes that remain (Equation 6.7).

Varied Percentages of Node Failure

We have demonstrated that higher scatter width leads to higher probability of data loss. This experiment studies data durability under varied percentages of nodes that fail in a failure event. There are five tests, wherein the number of nodes grows from 500 to 10000. In each test, the percentage of nodes that are selected to fail increases by 0.5% from 0 up to 5%. The scatter width is $S = 10$.

Figure 6.13 depicts the probability of data loss using ElasticCopyset. For a system of 500 nodes, the loss probability rises, but remains below 10% even when 5% of the nodes fail simultaneously. As the number of nodes increases, there is a clear trend that the probability of data loss grows more quickly. For example, when 5% of 10000 nodes fail concurrently, the probability of data loss is almost 90%. The reason is that, given the same percentage of node failure, there are more nodes failed in a system with more nodes, which naturally leads to higher data loss probability.

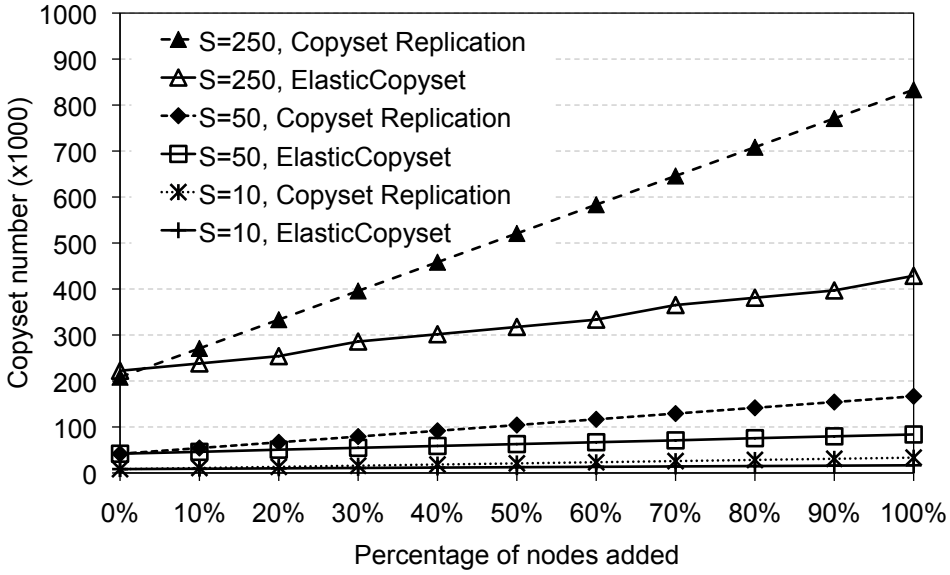
A horizontal line at 33.3% is added in Figure 6.13. If a system can tolerate one time of data loss in every three system failures, ElasticCopyset can withstand 5% of node failure rate for up to 2000 nodes. Alternatively, it can support a system of 10000 nodes, if no more than 3% of the nodes (i.e. 300 nodes) would fail simultaneously. We have also conducted this experiment using Copyset Replication. The results are almost identical to Figure 6.13, and are not shown so as to avoid repetition. A similar figure can be referred to Figure 8 in Cidon et al. (2013). The reason why Copyset Replication and ElasticCopyset exhibit a similar probability of data loss, is because both schemes generate a close to minimum number of copysets, as depicted in Figure 6.9.

6.4.2 Scaling at a Constant Rate

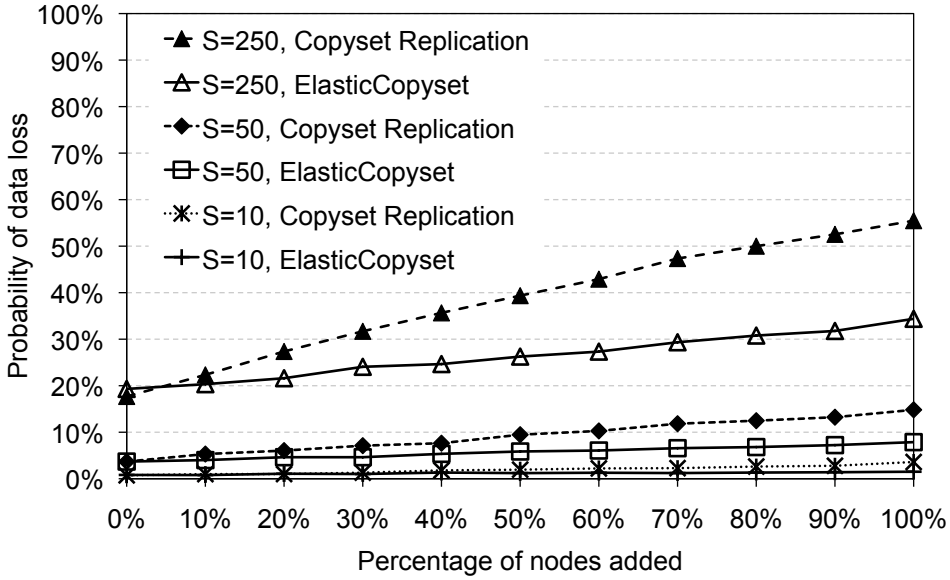
We have studied the data durability with static deployment, in which the copysets were statically generated based on the settings of the system. In this subsection, we study the scalability of the placement schemes with two online evaluations, wherein the number of nodes increases or decreases at a constant rate during runtime.

The first experiment is the evaluation on system scale-out. The placement schemes are required to handle node additions. The initial number of nodes is $N = 5000$. The new nodes are added one after another during the runtime, until the system scale is doubled (i.e., $N = 10000$). There are three tests, with the scatter width S set as 10, 50, and 250, respectively.

Figure 6.14a compares the number of copysets generated by Copyset Replication and ElasticCopyset during scale-out. In all the tests, as the number of nodes grows, the number of copysets increases more quickly when using Copyset Replication than when using ElasticCopyset. Consequently, as shown in Figure 6.14b, given 1% of nodes that fail simultaneously, ElasticCopyset consistently presents a smaller probability of data loss



(a) Number of copysets

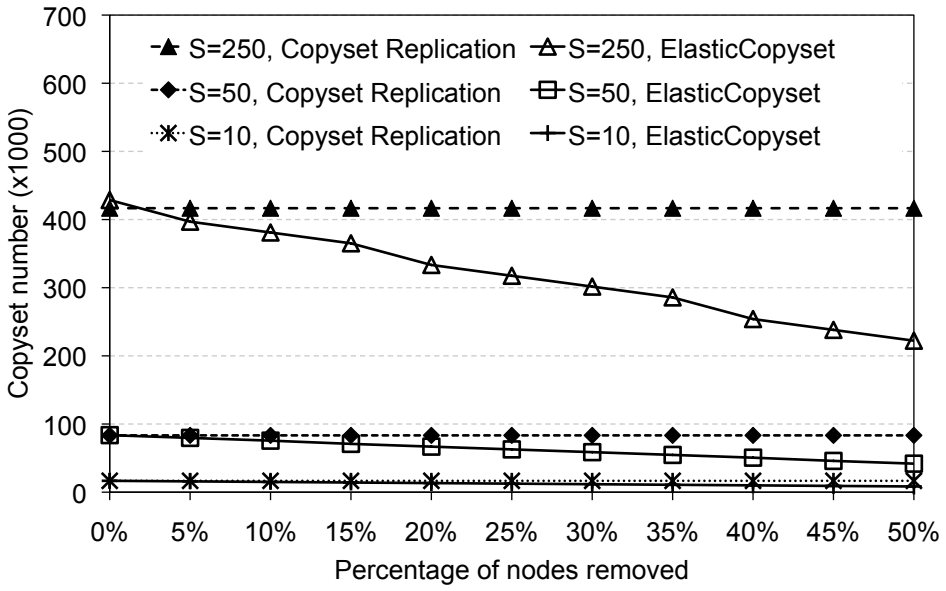


(b) Probability of data loss, with 1% of nodes selected to fail.

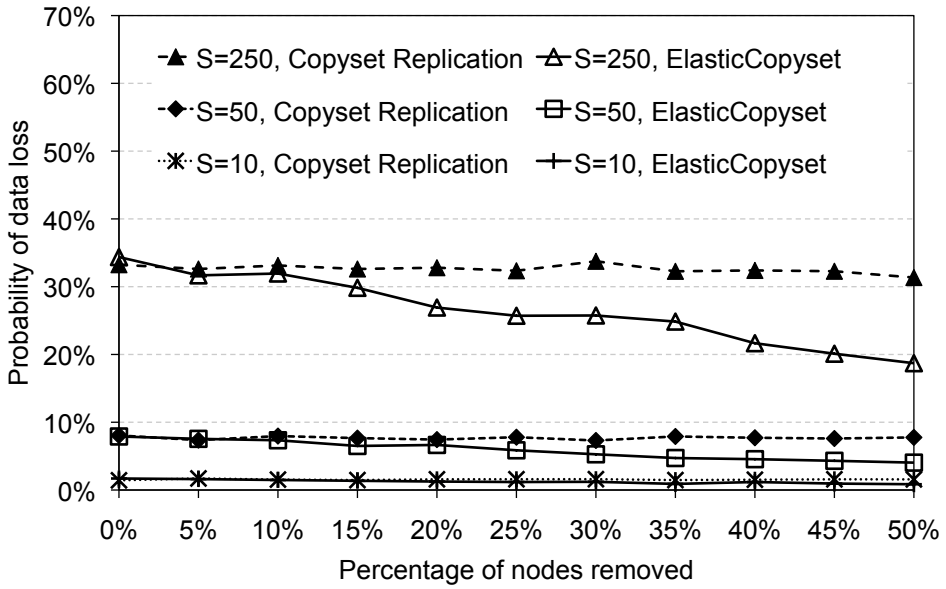
Figure 6.14: Scale out the system by adding nodes one-by-one at runtime. Given an initial number of nodes $N = 5000$ that rises up to 10000. The figures show varying percentage that the system is scaled, using varying setting of scatter width.

than Copyset Replication. When the system is scaled out by 100%, the probability of data loss in Copyset Replication is also almost twice the data loss probability in ElasticCopyset.

The second experiment is to study system scale-in, which is exactly the reverse of the scale-out experiment. The number of nodes is $N = 10000$. The nodes were removed from the system one-by-one during runtime, until the system scale was reduced to a half (i.e., $N = 5000$). Similarly, there are three tests, with $S = 10$, 50 and 250. The percentage of nodes that fail is also 1%.



(a) Number of copysets

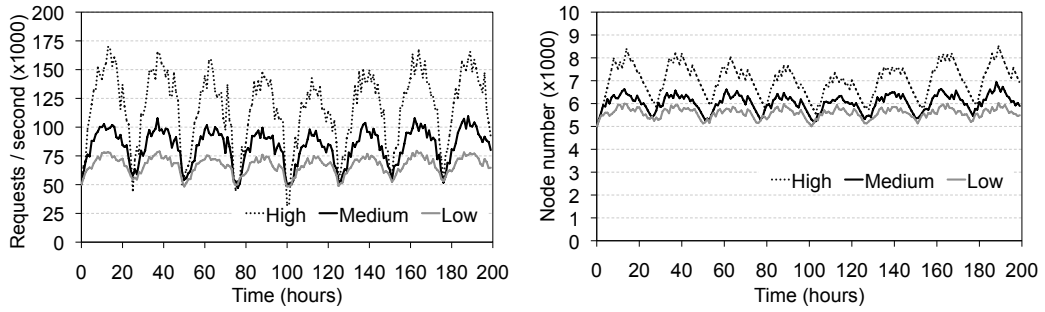


(b) Probability of data loss, with 1% of nodes selected to fail.

Figure 6.15: Scale in the system by removing nodes one-by-one at runtime. Given an initial number of nodes N=10000 that drops down to 5000. The figures show varying percentage that the system is scaled, using varying setting of scatter width.

As depicted in Figure 6.15a, as the system scales in from 10000 to 5000, the number of copysets remains constant when using Copyset Replication. In contrast, ElasticCopyset is able to steadily reduce the copyset number during the scale-in. As a result, ElasticCopyset exhibits a smaller probability of data loss than Copyset Replication in Figure 6.15b. When the system is scaled in by 50%, the probability of data loss in ElasticCopyset is 50% smaller than Copyset Replication.

The experiments have demonstrated that, ElasticCopyset handles online node addition



(a) Workload (b) Number of nodes

Figure 6.16: The input for testing the scalability of Copyset Replication and Elastic-Copyset under dynamic workloads.

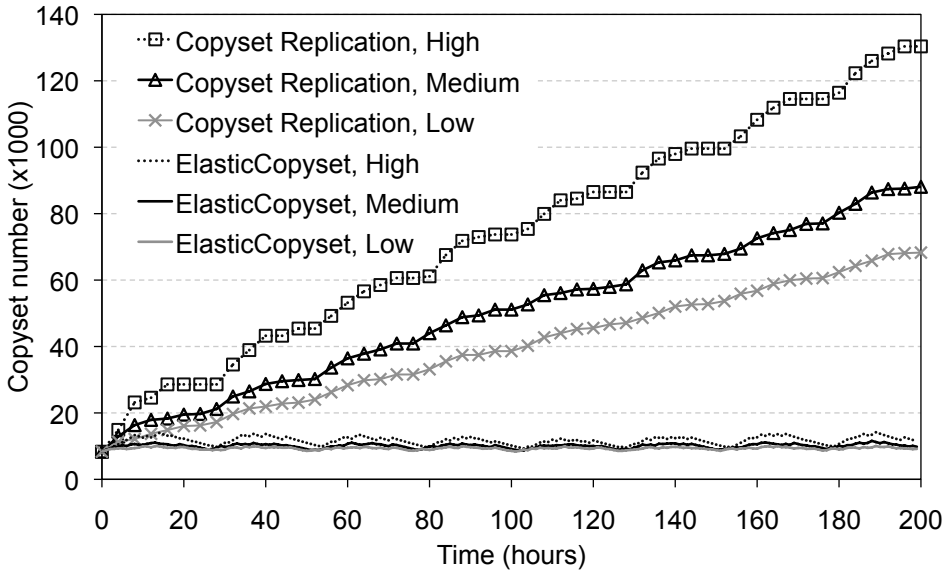
and removal better than Copyset Replication, and maintains a smaller probability of data loss than Copyset Replication during both scale-out and scale-in.

6.4.3 Elastic Scaling Under Workload

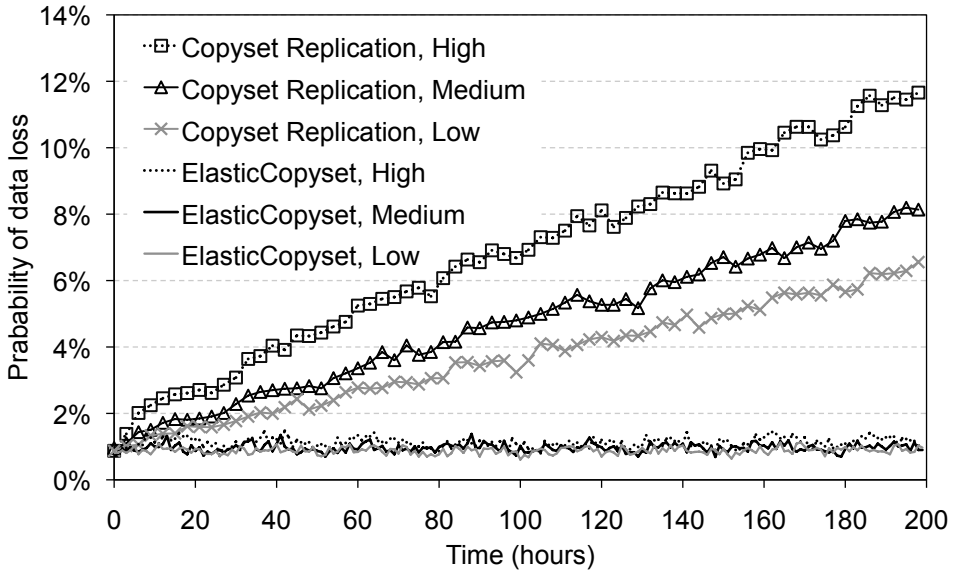
In this experiment, we study the scalability of the placement schemes when the system is subject to a dynamic workload. The test scenario is based on the workload analysis on Facebook's Memcached deployment (Atikoglu et al., 2012). The workload (e.g. number of requests per second) follows the diurnal pattern, wherein there are peaks in day time and bottoms in night time. The system is, either automatically or manually, scaled out and scaled in incrementally according to the changes of workload. During the process of dynamic scaling, we compared ElasticCopyset against Copyset Replication, on data durability.

The input of this experiment is shown in Figure 6.16. We simulated the workload to match the temporal patterns described in the Facebook deployment (Atikoglu et al., 2012). As depicted in Figure 6.16a, every 24 hours is a period of wave, and there are 200 hours in total. The bottom workload is set as 50,000 requests per second. Depending on the peak workload, there are three types of patterns, namely *low*, *medium*, and *high*. The peak workloads are 75000, 100000, and 150000 reqs/sec, respectively, which are 50%, 100% and 200% greater than the bottom workload. Hence, in the remaining figures of this subsection, the results are labeled as Low, Medium, and High, respectively.

Based on the three types of workload, we then simulated the changes in the number of nodes. As shown in Figure 6.16b, when the workload is at the bottoms, there are 5000 nodes. The number of nodes is increased as the workload rises up. The system scale peaks



(a) Numbers of copysets generated by ElasticCopyset and Copyset Replication



(b) Probability of data loss, given 1% of nodes selected to fail.

Figure 6.17: Compare the performance of dynamic scaling between Copyset Replication and ElasticCopyset. The system is scaled based on different types of workloads. Initial number of nodes is N=5000.

approximately at 6000, 7000, and 8500 nodes, for Low, Medium, and High Workload, respectively. Hence, given $N = 5000$ at the bottom, the percentages of scaling-out are respectively 20%, 40% and 80% under the workloads.

The number of nodes in Figure 6.16b is served as the direct input for evaluating the data durability at elastic scaling, the experimental results of which are presented in Figure 6.17. The numbers of copysets generated are shown in Figure 6.17a. Initially, about 8500 copysets were generated in both schemes. However, as the time elapses, the two schemes

exhibit two disparate trends. In Copyset Replication, the number of copysets cumulates step by step every 24 hours, and grows steadily each day. While in ElasticCopyset, the number of copysets rises and declines according to the number of nodes, and remains at a horizontal level in the long run.

Consequently, given 1% of nodes that fail, the probability of data loss, which is determined by the number of copysets, also exhibits two disparate trends. As shown in Figure 6.17b, the loss probability in Copyset Replication increases steadily day after day. During 200 hours, the probability grows from 1% to 12% under the high workload. At this rate, the probability of data loss will exceed 50% within 1000 hours (i.e. 40 days). In contrast, ElasticCopyset manages to maintain the data loss probability below 1.5% under varied workloads.

Figure 6.17 has demonstrated that, when the system is dynamically scaled out and scaled in within a range over a long period of time, ElasticCopyset is able to maintain data loss probability at a close to minimised level, while Copyset Replication is not.

6.4.4 Discussion

We have presented the experimental results. The evaluations on static deployment have demonstrated that, ElasticCopyset exhibits a close to minimised probability of data loss under varied system setups and failure rates. Yet, it is noticeable in Figure 6.10b that ElasticCopyset generates more copysets than Copyset Replication, resulting in slightly higher probability of data loss shown in Figure 6.10a. This is due to the extra copysets generated in the incomplete group (Equation 6.7). Nevertheless, as shown in Figure 6.9b, the percentage of extra copysets is less than 6%.

The evaluations on scalability have shown that ElasticCopyset greatly outperforms Copyset Replication by maintaining the minimum copysets during online node addition and removal. The results can be explained. During scale-out, Copyset Replication requires $\frac{S}{R-1}$ new copysets to be generated for each node addition, while ElasticCopyset generates new copysets only when a new incomplete group is required (Figure 6.6). To illustrate, we assume n new nodes are added. Copyset Replication creates $\frac{S}{R-1}n$ new copysets, while ElasticCopyset creates $\frac{S}{R-1}\frac{N_G}{R}\lceil\frac{n}{N_G}\rceil$ new copysets (Equation 6.6). When $n \gg N_G$, then $\frac{S}{R-1}\frac{N_G}{R}\lceil\frac{n}{N_G}\rceil \approx \frac{S}{R-1}\frac{n}{R}$. Since $\frac{S}{R-1}n$ is R times as many as $\frac{S}{R-1}\frac{n}{R}$, given $R = 3$, then the number of copysets generated for new nodes is almost tripled in Copyset Replication than

in ElasticCopyset.

During scale-in, when a node is removed, Copyset Replication uses an existing node to replace it, without destroying any existing copyset. While in ElasticCopyset, the whole incomplete group is dismissed when there is no *Extra* node (shown in Figure 6.7). Hence, when the system is scaled in, the number of copysets in ElasticCopyset is reduced accordingly, while the number in Copyset Replication remains unchanged.

Overall, ElasticCopyset leverages the incomplete group to create and dismiss copysets for dynamic node addition and removal, and also maintains a close to minimised probability of data loss in both static deployment and dynamic scaling.

6.5 Chapter Summary

This paper addressed the issue of data durability under simultaneous node failures, for distributed data stores that are required to efficiently scale out and scale in, in direct response to workload demands. We have proposed ElasticCopyset, a novel, general-purpose data placement scheme that builds on the concept of "copysets" (Cidon et al., 2013). Given the scatter width of the data placement, ElasticCopyset defines the minimum number of nodes that can form a group, and splits the nodes in the system into a list of complete group and one incomplete group. ElasticCopyset uses a novel shuffle algorithm to generate a minimum number of non-overlapping copysets within each group to minimise the probability of data loss. ElasticCopyset also leverages the incomplete group to efficiently handle node addition and removal, such that each node operation affects only one scatter width of nodes.

We have evaluated ElasticCopyset against the current-state data placement schemes including Random Replication and Copyset Replication. The evaluation has demonstrated that ElasticCopyset is able to maintain a close to minimum probability with the setting of varying values of scatter width, number of nodes, and percentage of node failure. In contrast to the current-state Copyset Replication, ElasticCopyset has also exhibited much better scalability and elasticity in the scenario where the distributed system is required to dynamically scale in and scale out under the diurnal workload pattern.

Chapter 7

Conclusions and Future Directions

Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience.

– Roger Bacon

The main contribution of this thesis lies in the set of data management schemes, which have now been proposed, elaborated and evaluated. This concluding chapter summarises the main ideas and contributions of this thesis, and then discusses the potential future research directions.

7.1 Thesis Summary

Over the past few years, Cloud computing has emerged as a hundred billion dollar industry and as a successful paradigm for delivering computing utility as a service. Irrespective of Cloud service models or Cloud abstractions, distributed data stores form a critical component in the stack of data-centric applications and services on the Cloud.

This thesis began by studying, characterising and categorising several aspects of distributed data stores. It then identified Key-Value Stores (KVSs) as the state-of-the-art data stores that are most applicable to the IaaS model of Cloud computing. KVSs have advantages of high availability and inherent scalability, the features that are lacking in RDBMS that focus on ACID transactions. Hence, compared to RDBMS, KVSs are considered to be more friendly to the IaaS Cloud. Nevertheless, resource elasticity, a key feature of the Cloud, requires the KVSs to efficiently incorporate and dismiss the provisioned resources as nodes of the KVS according to workload changes. This thesis has found

that current-state KVSs are not capable of doing so, mostly due to the lack of a set of data management schemes that focus on addressing the issues of dynamic node additions and removals.

Hence, this thesis has proposed the design of a data distribution middleware, which aims to help shared-nothing KVSs improve the efficiency of elasticity in several aspects. First, in order to reduce the overhead of data movement across nodes, it relies on an automated partitioning algorithm that divides the data set into multiple partitions, and then consolidates every partition replica into standalone transferable units with a bounded size. Second, to achieve a balanced load at each stable scale after node additions and removals, it uses a set of data placement algorithms to distribute the transferable partition replicas amongst the nodes, with the aims of balancing both workload demand and data volume. Third, to maintain data consistency during data movement, it proposes a token management policy that controls the qualification of the source and destination nodes regarding to serving reads and writes. Last but not least, to address the issue of distributed synchronisation and decision making without a dedicated component, this thesis proposes an election-based coordination to facilitate the execution of distributed tasks in a decentralised manner.

Moreover, in order to evaluate the efficiency of elasticity, this thesis has presented the implementation of ElasCass for the proposed data distribution middleware. ElasCass was built on an open-source KVS called Cassandra (Apache, 2009), which follows the decentralised, shared-nothing architecture. ElasCass differs from Cassandra in several ways. First, to facilitate dynamic node additions and removals with finer-grained load balancing, ElasCass supports an ongoing view of a many-to-many mapping between partition replicas and nodes. The ongoing view also helps to ensure data consistency during data movement. Second, ElasCass supports automated partitioning in a background process. The resulting partition replicas are standalone transferable units that can be placed at any node at anytime, which simplifies data movement to a large extent. Additionally, ElasCass gathers detailed load statistics for the reallocation of partition replicas.

Given the implementation of ElasCass, the evaluations were conducted on the Amazon EC2 public IaaS, using YCSB as the benchmark tool. The experimental results have demonstrated that, ElasCass is capable of incorporating new empty nodes consecutively in a relatively short time, with ideally balanced data distribution and well-balanced work-

load. Compared to Cassandra, ElasCass exhibits better scalability and higher resource utilisation in computing and storage, and delivers a query throughput more than twice that of Cassandra. Moreover, ElasCass reduces the time required to incorporate a new node from several hours to within ten minutes, hence fulfilling the goal of efficient node addition.

Furthermore, hardware failure on the IaaS Cloud usually causes multiple VMs to fail simultaneously. This thesis has proposed ElasticCopyset, to ensure high data availability whilst allowing dynamic node additions and removals. ElasticCopyset divides the nodes into groups, wherein the nodes are shuffled to form a minimum number of non-overlapping sets of nodes (i.e. copysets). ElasticCopyset extends the concept of copyset (Cidon et al., 2013) to produce a brand-new set of algorithms for replication under elasticity. This thesis has also presented mathematical proof for the correctness of the algorithms. The evaluation has demonstrated that ElasticCopyset is able to maintain a close to minimum probability with the setting of varying values of scatter width, number of nodes, and percentage of node failure. For example, given a cluster of 5000 nodes and 1% of node failure rate, ElasticCopyset managed to reduce the data loss probability from 99.99% to less than 3.8%. In contrast to the current-state copyset-based replication, ElasticCopyset has also exhibited much better scalability and elasticity in the scenario where the distributed system is required to dynamically scale in and scale out under the diurnal workload pattern.

Overall, this thesis has presented the design and implementation of a set of data management schemes to achieve lightweight data movement, better load balancing, and higher data durability, while allowing dynamic node additions and removals, which together improve the efficiency of elasticity for decentralised, shared-nothing KVSs that are deployed on the IaaS Cloud.

7.2 Future Directions

7.2.1 Supporting Richer Queries

The applicability of KVSs depends on the variety of query commands they provide. Current-state KVSs are not constrained by the simple CRUD (i.e. create, read, update, delete) operations. Instead, they need to provide a richer set of commands for query

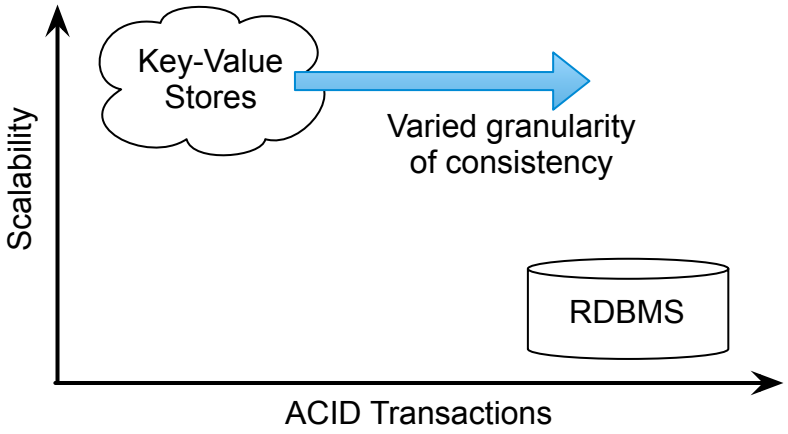


Figure 7.1: Varied consistency levels while maintaining scalability of Key-Value Stores

processing. There are two major directions, discussed as follows.

One direction is to support efficient range queries for the sake of online analytical processing (OLAP), catering for the application of business intelligence. Range query is important to ad-hoc analytical processing. The challenge of range query lies in maintaining cluster-wide load balancing while allocating consecutive key ranges within a small number of nodes. Note that the performance of range query depends on how quickly the query results can be returned in a sorted order, and therefore, given the same key ranges, a smaller number of nodes require less distributed synchronisation.

There have been research efforts on range queries. Vo et al. (2010) proposed to employ BATON, a balanced tree structure for overlay network (Jagadish et al., 2005), to effectively support load-adaptive data placement. It not only differentiates the master replica from slave replicas of each data partition, but also creates extra "secondary replicas" for load balancing. This approach is not applicable to the design of this thesis, which proposes to use copyset-based placement, wherein each data partition has the same number of indistinguishable replicas. In contrast, ES^2 (Cao et al., 2011) is a distributed data store that provides the implementation of an abstractive overlay based on the Cayley graph model, so as to support distributed indexing for efficient data retrieval in OLAP workloads. According to Lupu et al. (2008), the Cayley graph serves the algebraic and combinatorial base for many P2P networks, and provides a way to traverse all the nodes in an orderly fashion, so as to answer range queries efficiently. Hence, the future work is to employ the Cayley graph structure for the support of range query in copyset-based placement.

The other future direction is to enable ACID transactions for KVSs. This is motivated

by the lack of transactional support in existing commercial and open-source KVSs (DeCandia et al. 2007, Apache 2009), and the desire for traditional database applications, e.g., credit card transactions in online shopping. As shown in Figure 7.1, current-state KVSs are aiming to provide stronger data consistency for ACID transactions, without sacrificing the inherent scalability.

In the domain of elastic KVSs, this is achieved by supporting localised transaction. Bigtable (Chang et al., 2006) supports row-level transaction only, while Megastore (Baker et al., 2011) supports transactions over multiple records. Yet, Megastore requires data objects be partitioned into a collection of entity groups, wherein each entity must live within a single scope of serialisability, i.e. one machine or cluster (Helland, 2007). ElasTras (Das et al., 2009) also supports a restricted transaction semantics, termed as mini-transactions (Aguilera et al., 2007), which is executed within one data partition. However, it requires the data be statically partitioned, and is not applicable to KVSs that use automated partitioning as proposed in this thesis. Hence, there is a gap in supporting transactions over multiple data objects, without restricting data partitioning schemes.

7.2.2 Extending ElasticCopyset

Chapter 6 presented a placement algorithm called ElasticCopyset. It has been proved that, the copyset-based replication scheme is robust in terms of data durability at the event of simultaneous node failures, and the organisation of group/copyset division is efficient in supporting dynamic node arrivals and departures. Yet, there are two directions for extensions to this research in the near future.

One direction is to realise ElasticCopyset on a large-scale KVS. As discussed in Subsection 6.3.5 (page 165), the implementation of ElasticCopyset requires the realisation of several components:

- **Cluster initialisation**. The problem is how a cluster of nodes can spontaneously form groups and copysets without a dedicated component and with minimised human administration. In Subsection 6.3.5, we have proposed to designate one or a few coordinators before system startup. These coordinators recruit nodes as group members using Paxos-based coordination, and they continue to elect new coordinators to form more groups, until there is no node remaining. Within each group, each node is assigned with a group ID and a position in the group, which is propagated

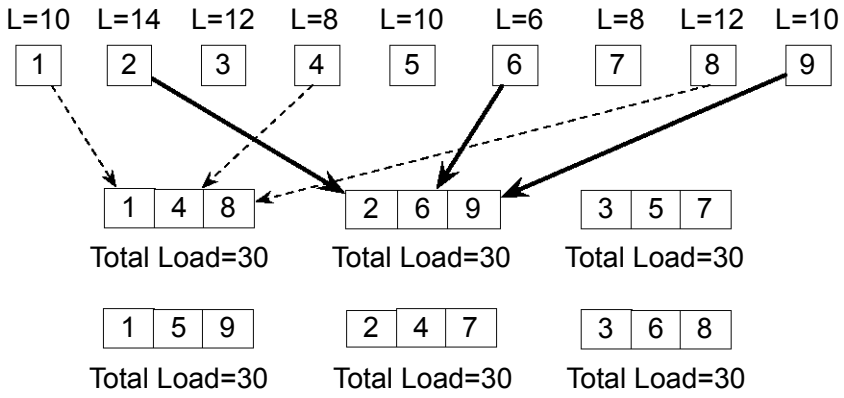


Figure 7.2: An example of balanced load over copysets with imbalanced load across nodes

to every other node, so that every node can calculate the copysets locally. Hence, copysets are formed without dedicated components.

- **Replica placement**. Subsection 4.2.3 (page 80) has proposed a set of algorithms to assign the replicas of certain data to each individual node. However, ElasticCopyset is a copyset-based scheme, wherein all the replicas of certain data should be assigned to the same copyset of nodes. It gives rise to the challenge of load balancing in ElasticCopyset, as the nodes within the same copyset can be varied greatly, even though the load is balanced over the copysets. Figure 7.2 presents such an example. There are nine nodes that form six copysets. Although the total load (either workload demand or data volume) of every copyset is the same (e.g. equals 30), the load in each individual node varies drastically from 6 to 14. Hence, there is a lack of replica placement strategies that achieve load balancing across nodes, while assigning replicas on a per-copyset basis.

- **Recovery of multiple node failures**. In Subsection 6.3.5, we have proposed to use the living Extra nodes in the incomplete group, to replace the failed nodes in the complete groups. However, when dealing with node replacement in the groups, there is still a challenge of balancing the load across nodes, which is determined by the replica placement strategies discussed. Hence, the replica placement problem for the copyset-based schemes shall be addressed in the future.

The other direction is to extend ElasticCopyset for the support of various replication numbers. In the current design, ElasticCopyset requires that the replication number is $R = 3$, that is, each data object (or partition) is replicated to three distinct nodes. This

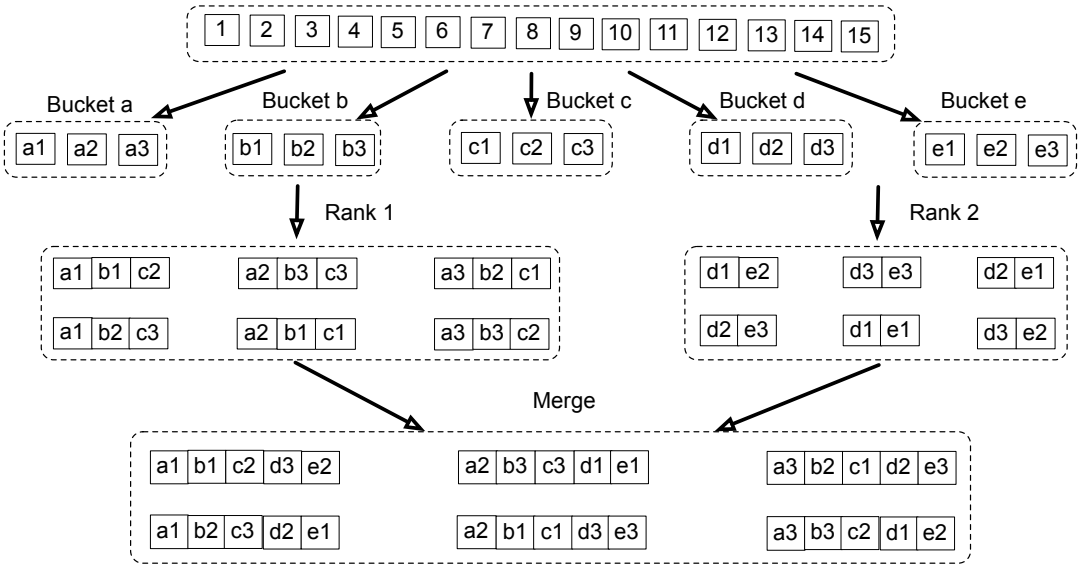


Figure 7.3: An extended shuffle algorithm for a higher replication number. In this example, $R = 5$ and $S = 8$. Hence, there are $\frac{S}{R-1} = 2$ rows, and $R = 5$ columns.

design choice follows many distributed data stores (Chang et al. 2008, Lakshman & Malik 2010, Ousterhout et al. 2010, Shvachko et al. 2010). Still, we intend to investigate the possibility of supporting a wider range of replication numbers.

The shuffle algorithm in ElasticCopyset uses three distinct orders to place the nodes (Figure 6.4 on page 156), so that any two resulting copysets overlap by at most one node. However, this problem is increasingly difficult with a higher replication number, since there are very limited distinct orders to shuffle nodes in a matrix. Still, there are ways to generate copysets for a higher replication number with a shuffle algorithm.

Figure 7.3 illustrates the intuition of such a shuffle algorithm. The basic idea is to split a higher replication number R into several smaller $R_i \leq 3$. For example, when $R = 5$, we split it into $R_1 = 3$ and $R_2 = 2$. To generate copysets, we still divide the group of nodes into $R = 5$ buckets. But, we use the shuffle algorithm to generate one rank of sub-copysets consisting of $R_1 = 3$ nodes for the first three buckets, and to generate the other rank of sub-copysets consisting of $R_2 = 2$ nodes for the remaining buckets. Next, we merge each sub-copyset of $R_1 = 3$ nodes with one sub-copyset of $R_2 = 2$ nodes. In this way, each resulting copyset contains $R = 5$ nodes. This method can be extended for the use of much higher replication numbers. For example, when $R = 10$, then it can be split into $10 = 3 + 3 + 3 + 1$. That is, four ranks of sub-copysets are generated separately, and then merged into one rank of copysets, each consisting of ten nodes.

However, this proposal leads to one challenge that needs to be addressed in the future.

When we merge two sub-copysets from different ranks, it requires another shuffle order that is distinct from the three default orders defined. With the lack of such order, there may exist two copysets that share more than one common node. For example, in Figure 7.3, $c2$ and $e2$ appear together in two copysets. For certain nodes, the actual scatter width will be slightly smaller than the given S. Nevertheless, as discussed in Section 6.2, the scatter width is typically set as a multiple of $(R-1)$, and therefore, a greater R results in a greater S. Hence, the value of the actual scatter width is still sufficiently large.

To sum up, we aim to extend ElasCass to build an elastic KVS that is adaptive to the IaaS Cloud in the long term. This KVS shall be able to adapt its capacity according to the change of workload demands, to efficiently support both OLAP queries and ACID transactions, and to maintain data durability in the event of multiple node failures.

References

Abadi, D., Madden, S. & Ferreira, M. 2006, 'Integrating compression and execution in column-oriented database systems', in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ACM, pp. 671–682.

Aberer, K. 2011, 'Peer-to-peer data management', *Synthesis Lectures on Data Management*, vol. 3, no. 2, pp. 1–150.

Abiteboul, S., Buneman, P. & Suciu, D. 2000, *Data on the Web: From relations to semistructured data and XML*, Morgan Kaufmann, Massachusetts, USA.

Agrawal, D., El Abbadi, A., Das, S. & Elmore, A. J. 2011, 'Database scalability, elasticity, and autonomy in the cloud', in *Database Systems for Advanced Applications*, Springer, pp. 2–15.

Aguilera, M. K., Merchant, A., Shah, M., Veitch, A. & Karamanolis, C. 2007, 'Sinfonia: A new paradigm for building scalable distributed systems', in *ACM SIGOPS Operating Systems Review*, 6, ACM, pp. 159–174.

Amazon 2006a, 'Amazon simple storage service (Amazon S3)', Accessed 1 June 2007, <http://aws.amazon.com/s3/>.

Amazon 2006b, 'Amazon Elastic Compute Cloud (EC2): Scalable Cloud Servers', Accessed 1 December 2012, <http://aws.amazon.com/ec2/>.

Amazon 2007, 'Amazon Elastic Block Store (EBS)', Accessed 1 December 2012, <http://aws.amazon.com/ebs/>.

Amazon 2008, 'Amazon SimpleDB: Simple database service', Accessed 09 September 2013, <http://aws.amazon.com/simpledb/>.

Amazon 2011, 'Amazon elastic cloud computing instance types', Accessed 1 December 2012, <http://aws.amazon.com/ec2/instance-types/>.

Amazon 2012, 'Amazon DynamoDB: NoSQL database service', Accessed 13 September 2013, <http://aws.amazon.com/dynamodb/>.

Anand, A., Muthukrishnan, C., Kappes, S., Akella, A. & Nath, S. 2010, 'Cheap and Large CAMs for High Performance Data-Intensive Networked Systems.', in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 10, USENIX Association, San Jose, CA, USA, pp. 29–29.

Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L. & Vasudevan, V. 2009, 'FAWN: A fast array of wimpy nodes', in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, ACM, Big Sky, Montana, USA, pp. 1–14.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M. & Werthimer, D. 2002, 'SETI@Home: An experiment in public-resource computing', *Communications of the ACM*, vol. 45, no. 11, pp. 56–61.

Anderson, E., Spence, S., Swaminathan, R., Kallahalla, M. & Wang, Q. 2005, 'Quickly finding near-optimal storage designs', *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 4, pp. 337–374.

Androutsellis-Theotokis, S. & Spinellis, D. 2004, 'A survey of peer-to-peer content distribution technologies', *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 335–371.

Apache 2008a, 'Apache CouchDB', Accessed 9 July 2008, <http://couchdb.apache.org>.

Apache 2008b, 'The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing', Accessed 19 July 2010, <http://hadoop.apache.org/>.

Apache 2009, 'The Apache Cassandra Project', Accessed 17 February 2010, <http://cassandra.apache.org/>.

Apache 2010, 'Apache HBase is the Hadoop database, a distributed, scalable, big data store', Accessed 31 July 2010, <http://hbase.apache.org/>.

Apache 2011, 'The Apache Accumulo sorted, distributed key/value store is a robust, scalable, high performance data storage and retrieval system', Accessed 12 July 2013, <http://accumulo.apache.org/>.

Apache 2012a, 'Apache Avro is a data serialization system', Accessed 29 July 2013, <http://avro.apache.org/>.

Apache 2012b, 'The Apache Thrift software framework', Accessed 03 August 2013, <http://thrift.apache.org/>.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. et al. 2010, 'A view of cloud computing', *Communications of the ACM*, vol. 53, no. 4, pp. 50–58.

Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S. & Paleczny, M. 2012, 'Workload analysis of a large-scale key-value store', *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, pp. 53–64.

Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Léon, J.-M., Li, Y., Lloyd, A. & Yushprakh, V. 2011, 'Megastore: Providing scalable, highly available storage for interactive services', in *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, vol. 11, Asilomar, California, USA, pp. 223–234.

Barahmand, S. & Ghandeharizadeh, S. 2013, 'BG: A Benchmark to Evaluate Interactive Social Networking Actions.', in *6th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA, pp. 118–130.

Barker, S., Chi, Y., Moon, H. J., Hacigümüş, H. & Shenoy, P. 2012, 'Cut me some slack: Latency-aware live migration for databases', in *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, ACM, Berlin, Germany, pp. 432–443.

Basho Tech. 2012, 'Riak is an open source, distributed database', Accessed 03 May 2013, <http://basho.com/riak/>.

Beaver, D., Kumar, S., Li, H. C., Sobel, J., Vajgel, P. et al. 2010, 'Finding a needle in haystack: Facebook's photo storage', in *Proceedings of the 9th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI), vol. 10, USENIX Association, Vancouver, BC, Canada, pp. 1–8.

Bermbach, D. & Kuhlenkamp, J. 2013, 'Consistency in distributed storage systems: An overview of models, metrics and measurement approaches', in *The International Conference on NETworked sYStems (NETYS)*, Springer, Marrakesh, Marocco, pp. 175–189.

Bermbach, D. & Tai, S. 2011, 'Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior', in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, ACM, p. 1.

Berners-Lee, T., Hendler, J., Lassila, O. et al. 2001, 'The semantic web', *Scientific american*, vol. 284, no. 5, pp. 28–37.

Bernstein, P. A. & Goodman, N. 1981, 'Concurrency control in distributed database systems', *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221.

Bernstein, P. A., Hadzilacos, V. & Goodman, N. 1987, *Concurrency control and recovery in database systems*, vol. 370, Addison-Wesley New York.

Bharambe, A. R., Agrawal, M. & Seshan, S. 2004, 'Mercury: Supporting scalable multi-attribute range queries', *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 353–366.

Bloom, B. H. 1970, 'Space/time trade-offs in hash coding with allowable errors', *Communications of the ACM*, vol. 13, no. 7, pp. 422–426.

Bondy, J. A. & Murty, U. S. R. 1976, *Graph theory with applications*, vol. 290, Macmillan Publishers Ltd., London, UK.

Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A. & Rash, S. 2011, 'Apache Hadoop goes real-time at Facebook', in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ACM, Athens, Greece, pp. 1071–1080.

Burrows, M. 2006, 'The Chubby lock service for loosely-coupled distributed systems', in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, Seattle, WA, USA, pp. 335–350.

Buyya, R. & Vazhkudai, S. 2001, 'Compute power market: Towards a market-oriented grid', in *Proceedings of 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, Brisbane, Australia, pp. 574–581.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J. & Brandic, I. 2009, 'Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility', *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616.

Cao, Y., Chen, C., Guo, F., Jiang, D., Lin, Y., Ooi, B. C., Vo, H. T., Wu, S. & Xu, Q. 2011, 'ES^2: A cloud data storage system for supporting both OLTP and OLAP', in *Proceedings of the 27th International Conference on Data Engineering 2011 (ICDE)*, IEEE, Hannover, Germany, pp. 291–302.

Cattell, R. 2011, 'Scalable SQL and NoSQL data stores', *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27.

Ceri, S. & Pelagatti, G. 1984, *Distributed databases principles and systems*, McGraw-Hill Inc., New York, NY, USA.

Chandra, T., Griesemer, R. & Redstone, J. 2007, 'Paxos made live: An engineering perspective', in *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, vol. 7, Portland, Oregon, USA, pp. 398–407.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. & Gruber, R. 2006, 'Bigtable: A distributed structured data storage system', in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, Seattle, WA, USA, pp. 305–314.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. & Gruber, R. 2008, 'Bigtable: A distributed storage system for structured data', *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26.

Chang, F. W., Ji, M., Leung, S.-T., MacCormick, J., Perl, S. E. & Zhang, L. 2002, 'Myriad: Cost-effective disaster tolerance', in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, vol. 2, USENIX Association, Monterey, CA, USA, p. 8.

Chansler, R. J. 2012, 'Data availability and durability with the Hadoop distributed file system', *The USENIX Magazine, FILESYSTEMS*, vol. 37, no. 1, pp. 16–22.

Chodorow, K. & Dirolf, M. 2010, *MongoDB: The definitive guide*, 1st ed., O'Reilly Media Inc., CA, USA.

Chun, B.-G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M. F., Kubiatowicz, J. & Morris, R. 2006, 'Efficient replica maintenance for distributed storage systems', in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 6, USENIX Association, San Jose, CA, USA, pp. 4–4.

Cidon, A., Nagaraj, K., Katti, S. & Viswanath, P. 2012, 'MinCopysets: Derandomizing replication in cloud storage', Tech. rep., Stanford University, CA, USA. Accessed 30 June 2013, <http://www.stanford.edu/~cidon/materials/MinCopysets.pdf>.

Cidon, A., Rumble, S., Stutsman, R., Katti, S., Ousterhout, J. & Rosenblum, M. 2013, 'Copysets: Reducing the frequency of data loss in cloud storage', in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)*, USENIX Association, San Jose, CA, USA, pp. 37–48.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. & Warfield, A. 2005, 'Live migration of virtual machines', in *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, USENIX Association, Boston, MA, USA, pp. 273–286.

Clarke, I., Sandberg, O., Wiley, B. & Hong, T. W. 2001, 'Freenet: A distributed anonymous information storage and retrieval system', in *Designing Privacy Enhancing Technologies, Proceedings of International Workshop on Design Issues in Anonymity and Unobservability*, Springer, Berkeley, CA, USA, pp. 46–66.

Cockroft, A. 2011, 'Netflix goes global', in *14th International Workshop on High Performance Transaction Systems (HPTS)*, USENIX Association, Pacific Grove, CA, USA, pp. 80–80.

Codd, E. F. 1970, 'A relational model of data for large shared data banks', *Communications of the ACM*, vol. 13, no. 6, pp. 377–387.

Cohen, B. 2003, 'Incentives build robustness in BitTorrent', in *1st Workshop on Economics of Peer-to-Peer Systems*, vol. 6, Berkeley, CA, USA, pp. 68–72.

Cooper, B., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D. & Yerneni, R. 2008, 'PNUTS: Yahoo!'s hosted data serving platform', *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 2, pp. 1277–1288.

Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R. & Sears, R. 2010, 'Benchmarking cloud serving systems with YCSB', in *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, ACM, Indianapolis, IN, USA, pp. 143–154.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C. & Hochschild, P. 2012, 'Spanner: Google's globally-distributed database', in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 1, USENIX Association, Hollywood, CA, USA, pp. 251–264.

Couchbase 2012, 'Couchbase: Document-oriented NoSQL database', Accessed 12 December 2012, `<http://www.couchbase.com/>`.

Curino, C., Jones, E., Zhang, Y., Wu, E. & Madden, S. 2010a, 'Relational cloud: The case for a database service', Tech. Rep. MIT-CSAIL-TR-2010-014, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA.

Curino, C., Jones, E., Zhang, Y. & Madden, S. 2010b, 'Schism: A workload-driven approach to database replication and partitioning', *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 48–57.

Curino, C., Jones, E., Popa, R., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N. et al. 2011, 'Relational cloud: A database-as-a-service for the cloud', in *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA, pp. 235–240.

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R. & Stoica, I. 2001, 'Wide-area cooperative storage with CFS', *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 202–215.

Danga Interactive 2004, 'Memcached: A distributed memory object caching system', Accessed 03 September 2010, <http://www.memcached.org/>.

Das, S., Agrawal, D. & El Abbadi, A. 2009, 'ElasTraS: An elastic transactional data store in the cloud', in *Workshop on Hot Topics in Cloud Computing (HotCloud'09)*, 7, USENIX Association, San Diego, CA, USA, pp. 35–40.

Das, S., Agrawal, D. & El Abbadi, A. 2010a, 'G-Store: A scalable data store for transactional multi key access in the cloud', in *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, ACM, Indianapolis, IN, USA, pp. 163–174.

Das, S., Nishimura, S., Agrawal, D. & El Abbadi, A. 2010b, 'Live database migration for elasticity in a multitenant database for cloud platforms', Tech. Rep. 2010-09, CS, UCSB, Santa Barbara, CA, USA.

Das, S., Nishimura, S., Agrawal, D. & El Abbadi, A. 2011, 'Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration', *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, no. 8, pp. 494–505.

Das, S., Agrawal, D. & El Abbadi, A. 2013, 'ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud', *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 1, p. 5.

DataStax 2012, 'The data consistency model in Apache Cassandra 1.0', Accessed 12 December 2012, <http://www.datastax.com/docs/1.0/dml/data_consistency>.

DataStax 2013a, 'DataStax Enterprise: The enterprise NoSQL database platform for today's online applications', Accessed 23 August 2013, <http://www.datastax.com/what-we-offer/products-services/datastax-enterprise>.

DataStax 2013b, 'CQL for Cassandra 2.0', Accessed 04 December 2013, <http://www.datastax.com/documentation/cql/3.1/webhelp/index.html>.

DB-Engines 2013, 'DB-Engines ranking per database model category', Accessed 31 December 2013, <http://db-engines.com/en/ranking_categories>.

DB-Engines 2014a, 'Cassandra System Properties', Accessed 08 January 2014, <http://db-engines.com/en/system/Cassandra>.

DB-Engines 2014b, 'Redis System Properties', Accessed 08 January 2014, <http://db-engines.com/en/system/Redis>.

DB-Engines 2014c, 'DB-Engines ranking per database model category', Accessed 1 February 2014, <http://db-engines.com/en/ranking>.

Dean, J. & Ghemawat, S. 2008, 'MapReduce: Simplified data processing on large clusters', *Communications of the ACM*, vol. 51, no. 1, pp. 107–113.

Debnath, B., Sengupta, S. & Li, J. 2010a, 'ChunkStash: Speeding up inline storage deduplication using flash memory', in *Proceedings of the 2010 USENIX Annual Technical Conference (ATC'10)*, USENIX Association, Boston, MA, USA, pp. 16–16.

Debnath, B., Sengupta, S. & Li, J. 2010b, 'FlashStore: high throughput persistent key-value store', *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 1414–1425.

Debnath, B., Sengupta, S. & Li, J. 2011, 'SkimpyStash: RAM space skimpy key-value store on flash-based storage', in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ACM, Athens, Greece, pp. 25–36.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. 2007, 'Dynamo: Amazon's highly available key-value store', in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, vol. 7, ACM, Stevenson, WA, USA, pp. 205–220.

DeWitt, D. & Gray, J. 1992, 'Parallel database systems: the future of high performance database systems', *Communications of the ACM*, vol. 35, no. 6, pp. 85–98.

Ding, C. H., Nutanong, S. & Buyya, R. 2005, *Peer-to-peer computing: The Evolution of a Disruptive Technology*, chap. Peer-to-Peer Networks for Content Sharing, Igi Global, pp. 28–65.

Domingos, P. & Hulten, G. 2000, 'Mining high-speed data streams', in *Proceedings of the 6th ACM International Conference on Knowledge Discovery and Data mining (SIGKDD)*, ACM, pp. 71–80.

Dong, Y. 2009, 'Hypertable goes realtime at Baidu', Accessed 19 December 2013, <http://download.hypertable.com/pub/hypertable-goes-realtime-at-baidu.pdf>.

Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P. & Neugebauer, R. 2003, 'Xen and the art of virtualization', in *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, ACM, New York, NY, USA, pp. 164–177.

Elmore, A. J., Das, S., Agrawal, D. & El Abbadi, A. 2011, 'Zephyr: Live migration in shared nothing databases for elastic cloud platforms', in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ACM, pp. 301–312.

Escriva, R., Wong, B. & Sirer, E. G. 2012, 'HyperDex: A distributed, searchable key-value store', in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Helsinki, Finland, pp. 25–36.

Expósito, R. R., Taboada, G. L., Ramos, S., González-Domínguez, J., Touriño, J. & Doallo, R. 2013, 'Analysis of I/O performance on an amazon EC2 cluster compute and high I/O platform', *Journal of grid computing*, vol. 11, no. 4, pp. 613–631.

FastTrack 2001, 'FastTrack Peer-to-Peer Technology Company', Accessed 20 March 2001, `<http://www.fasttrack.nu/>`.

Fitzpatrick, B. 2004, 'Distributed caching with memcached', *Linux journal*, vol. 2004, no. 124, p. 5.

Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C. & Quinlan, S. 2010, 'Availability in globally distributed storage systems', in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, Vancouver, BC, Canada, pp. 61–74.

Foster, I., Zhao, Y., Raicu, I. & Lu, S. 2008, 'Cloud computing and grid computing 360-degree compared', in *Grid Computing Environments Workshop (GCE'08)*, IEEE, pp. 1–10.

Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A. & Gauthier, P. 1997, 'Cluster-based scalable network services', in *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, ACM, New York, NY, USA, pp. 78–91.

Gartner, Inc. 2013, 'Gartner says Cloud computing will become the bulk of new IT spend', Accessed 12 December 2013, <http://www.gartner.com/newsroom/id/2613015>.

George, L. 2011, *HBase: The definitive guide*, O'Reilly Media Inc., CA, USA.

Ghemawat, S., Gobioff, H. & Leung, S. 2003, 'The Google file system', in *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, ACM, New York, NY, USA, pp. 29–43.

Gibson, G. A. & Van Meter, R. 2000, 'Network attached storage architecture', *Communications of the ACM*, vol. 43, no. 11, pp. 37–45.

Gifford, D. K. 1981, 'Information storage in a decentralized computer system', Tech. Rep. CSL-81-8, Xerox PARC.

Gilbert, S. & Lynch, N. 2002, 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services', *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59.

GitHub.com 2010, 'BrianFrankCooper/YCSB at GitHub', Accessed 04 May 2013, <https://github.com/brianfrankcooper/YCSB>.

Gnutella 2000, 'The Gnutella Web caching system', Accessed 14 March 2000, <http://www.gnucleus.com/gwebcache/>.

Google 2008, 'Google App Engine: Platform as a Service', Accessed 3 December 2011, <https://developers.google.com/appengine/>.

Gray, J. 1992, *Benchmark handbook: for database and transaction processing systems*, Morgan Kaufmann Publishers Inc., Burlington, Massachusetts, USA.

Gray, J. et al. 1981, 'The transaction concept: Virtues and limitations', in *VLDB*, vol. 81, pp. 144–154.

Guo, Z., McDirmid, S., Yang, M., Zhuang, L., Zhang, P., Luo, Y., Bergan, T., Bodik, P., Musuvathi, M. & Zhang, Z. 2013, 'Failure recovery: When the cure is worse than the disease', in *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems (HotOS'13)*, USENIX Association, Santa Ana Pueblo, New Mexico, USA, pp. 8–8.

Gupta, A., Agrawal, D. & El Abbadi, A. 2003a, 'Approximate Range Selection Queries in Peer-to-Peer Systems.', in *CIDR*, vol. 3, pp. 141–151.

Gupta, A., Liskov, B. & Rodrigues, R. 2003b, 'One hop lookups for peer-to-peer overlays', in *Proceedings of the 9th USENIX conference on Hot Topics in Operating Systems (HotOS'03)*, USENIX Association, Lihue (Kauai), Hawaii, USA, pp. 7–12.

Hacigumus, H., Tatemura, J., Hsiung, W.-P., Moon, H. J., Po, O., Sawires, A., Chi, Y. & Jafarpour, H. 2010, 'CloudDB: One size fits all revived', in *6th World Congress on Services (SERVICES)*, IEEE, pp. 148–149.

Haeberlen, A., Mislove, A. & Druschel, P. 2005, 'Glacier: Highly durable, decentralized storage despite massive correlated failures', in *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 2, USENIX Association, Boston, MA, USA, pp. 143–158.

Hamilton, J. 2008, 'Geo-replication at facebook', Accessed 1 June 2011, `<http://perspectives.mvdirona.com/2008/08/21/GeoReplicationAtFacebook.aspx>`.

Harvey, N. J., Jones, M. B., Saroiu, S., Theimer, M. & Wolman, A. 2003, 'SkipNet: A scalable overlay network with practical locality properties', in *4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, vol. 274, USENIX Association, Seattle, WA, USA, pp. 113–126.

Hazelcast 2013, 'Hazelcast: The leading open source in-memory data grid', Accessed 20 December 2013, `<http://www.hazelcast.org/>`.

Helland, P. 2007, 'Life beyond distributed transactions: An apostate's opinion', in *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, pp. 132–141.

Herbst, N. R., Kounev, S. & Reussner, R. 2013, 'Elasticity in Cloud Computing: What It Is, and What It Is Not', in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, USENIX, San Jose, CA, USA, pp. 23–27.

Herlihy, M. P. & Wing, J. M. 1990, 'Linearizability: A correctness condition for concurrent objects', *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492.

Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S. & Stoica, I. 2003, 'Querying the Internet with PIER', in *Proceedings of the 29th international conference on Very large data bases (VLDB'03)*, vol. 29, VLDB Endowment, Berlin, Germany, pp. 321–332.

Hunt, P., Konar, M., Junqueira, F. P. & Reed, B. 2010, 'ZooKeeper: Wait-free coordination for internet-scale systems', in *Proceedings of the 2010 USENIX Annual Technical Conference (ATC'10)*, vol. 8, USENIX Association, Boston, MA, USA, pp. 11–11.

Hypertable 2009, 'Hypertable: Big data, big performance', Accessed 09 May 2013, `<http://hypertable.com/documentation/architecture/>`.

Jagadish, H. V., Ooi, B. C. & Vu, Q. H. 2005, 'Baton: A balanced tree structure for peer-to-peer networks', in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, VLDB Endowment, Trondheim, Norway, pp. 661–672.

Kaashoek, M. F. & Karger, D. R. 2003, 'Koorde: A simple degree-optimal distributed hash table', in *Peer-to-Peer Systems II*, Springer, pp. 98–107.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. & Lewin, D. 1997, 'Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web', in *Proceedings of the 29th Annual Symposium on Theory of Computing (STOC'97)*, ACM, El Paso, TX, USA, pp. 654–663.

Kavalanekar, S., Worthington, B., Zhang, Q. & Sharda, V. 2008, 'Characterization of storage workload traces from production windows servers', in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, Seattle, Washington, USA, pp. 119–128.

Kazaa 2001, 'Kazaa Media Desktop', Accessed 31 March 2001, `<http://www.kazaa.com>`.

Kivity, A., Kamay, Y., Laor, D., Lublin, U. & Liguori, A. 2007, 'KVM: the Linux virtual machine monitor', in *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230.

Klyne, G., Carroll, J. J. & McBride, B. 2004, 'Resource description framework (RDF): Concepts and abstract syntax', *W3C recommendation*, vol. 10.

Koh, Y., Knauerhase, R., Brett, P., Bowman, M., Wen, Z. & Pu, C. 2007, 'An analysis of performance interference effects in virtual environments', in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS 2007)*, IEEE, pp. 200–209.

Krishnan, P., Raz, D. & Shavitt, Y. 2000, 'The cache location problem', *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 5, pp. 568–582.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W. et al. 2000, 'Oceanstore: An architecture for global-scale persistent storage', *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201.

Lakshman, A. & Malik, P. 2010, 'Cassandra: A decentralized structured storage system', *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40.

Lamb, C., Landis, G., Orenstein, J. & Weinreb, D. 1991, 'The ObjectStore database system', *Communications of the ACM*, vol. 34, no. 10, pp. 50–63.

Lamport, L. 1978, 'Time, clocks, and the ordering of events in a distributed system', *Communications of the ACM*, vol. 21, no. 7, pp. 558–565.

Lamport, L. 1998, 'The part-time parliament', *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169.

Lamport, L. 2001, 'Paxos made simple', *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25.

Laoutaris, N., Telelis, O., Zissimopoulos, V. & Stavrakakis, I. 2006, 'Distributed selfish replication', *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 17, no. 12, pp. 1401–1413.

Lee, E. K. & Thekkath, C. A. 1996, 'Petal: Distributed virtual disks', *ACM SIGOPS Operating Systems Review*, vol. 30, pp. 84–92.

Leventhal, A. 2008, 'Flash storage memory', *Communications of the ACM*, vol. 51, no. 7.

Li, A., Yang, X., Kandula, S. & Zhang, M. 2010, 'CloudCmp: Comparing public cloud providers', in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, ACM, pp. 1–14.

Lim, H., Fan, B., Andersen, D. G. & Kaminsky, M. 2011, 'SILT: A memory-efficient, high-performance key-value store', in *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, ACM, Cascais, Portugal, pp. 1–13.

Lim, H. C., Babu, S. & Chase, J. S. 2010, 'Automated control for elastic storage', in *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)*, IEEE, Washington, DC, USA, pp. 1–10.

Lin, M. J. & Marzullo, K. 1999, 'Directional gossip: Gossip in a wide area network', in *Proceedings of the 3rd European Dependable Computing Conference (EDCC)*, Springer, Prague, Czech Republic, pp. 364–379.

Litwin, W. 1980, 'Linear hashing: A new tool for file and table addressing', in *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB)*, vol. 80, IEEE Computer Society, Montreal, Canada, pp. 1–3.

Lua, E. K., Crowcroft, J., Pias, M., Sharma, R. & Lim, S. 2005, 'A survey and comparison of peer-to-peer overlay network schemes.', *IEEE Communications Surveys and Tutorials*, vol. 7, no. 1-4, pp. 72–93.

Lupu, M., Ooi, B. C. & Tay, Y. 2008, 'Paths to stardom: calibrating the potential of a peer-based data management system', in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ACM, Vancouver, BC, Canada, pp. 265–278.

Malkhi, D., Naor, M. & Ratajczak, D. 2002, 'Viceroy: A scalable and dynamic emulation of the butterfly', in *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, ACM, Monterey, CA, USA, pp. 183–192.

Maymounkov, P. & Mazieres, D. 2002, 'Kademlia: A peer-to-peer information system based on the xor metric', in *Peer-to-Peer Systems*, Springer, pp. 53–65.

Mell, P. & Grance, T. 2011, 'The NIST definition of cloud computing', *NIST special publication*, vol. 800, no. 145, p. 7.

MemcacheDB 2008, 'MemcacheDB: A distributed key-value storage system designed for persistent', Accessed 20 January 2009, `<http://memcachedb.org/>`.

Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S. & Xu, Z. 2002, 'Peer-to-peer computing', Tech. Rep. HPL-2002-57, HP Lab, Palo Alto, CA, USA.

Moulavi, M. A., Al-Shishtawy, A. & Vlassov, V. 2012, 'State-space feedback control for elastic distributed storage in a cloud environment', in *The 8th International Conference on Autonomic and Autonomous Systems (ICAS)*, St. Maarten, Netherlands Antilles, pp. 18–27.

Nakamoto, S. 2008, 'Bitcoin: A peer-to-peer electronic cash system', Accessed 1 March 2012, <http://bitcoin.org/bitcoin.pdf>.

Napster 1999, 'Napster Online Music Store', Accessed 30 June 1999, <http://www.napster.com>.

Narayanan, D., Donnelly, A., Thereska, E., Elnikety, S. & Rowstron, A. I. 2008, 'Everest: Scaling down peak loads through I/O off-loading', in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, USENIX Association, San Diego, CA, USA, pp. 15–28.

Nathuji, R., Kansal, A. & Ghaffarkhah, A. 2010, 'Q-clouds: managing performance interference effects for QoS-aware clouds', in *Proceedings of the 5th European conference on Computer systems*, ACM, pp. 237–250.

Neo Technology 2010, 'Neo4j: The world's leading graph database', Accessed 28 March 2010, <http://www.neo4j.org/>.

Objectivity Inc. 2010, 'InfiniteGraph', Accessed 21 December 2011, <http://www.objectivity.com/infinitegraph>.

Olston, C., Reed, B., Srivastava, U., Kumar, R. & Tomkins, A. 2008, 'Pig latin: A not-so-foreign language for data processing', in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ACM, Vancouver, BC, Canada, pp. 1099–1110.

O'Neil, P., Cheng, E., Gawlick, D. & O'Neil, E. 1996, 'The log-structured merge-tree (LSM-tree)', *Acta Informatica*, vol. 33, no. 4, pp. 351–385.

Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J. & Rosenblum, M. 2011, 'Fast crash recovery in RAMCloud', in *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, ACM, Cascais, Portugal, pp. 29–41.

Oracle 2001, 'Oracle Real Application Clusters (RAC) database', Accessed 03 September 2011, `<http://www.oracle.com/us/products/database/options/real-application-clusters/overview/index.html>`.

Oram, A. 2001, *Peer-to-peer: Harnessing the Power of a Disruptive Technologies*, O'Reilly Media Inc., CA, USA.

Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M. et al. 2010, 'The case for RAMClouds: Scalable high-performance storage entirely in DRAM', *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105.

Pagh, R. & Rodler, F. F. 2004, 'Cuckoo hashing', *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144.

Paiva, J., Ruivo, P., Romano, P. & Rodrigues, L. 2013, 'AUTOPLACER: Scalable self-tuning data placement in distributed key-value stores', in *10th International Conference on Autonomic Computing (ICAC)*, USENIX Association, San Jose, CA, USA, pp. 119–131.

Papadimitriou, C. H. 1979, 'The serializability of concurrent database updates', *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653.

Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A. & Rinaldi, B. 2011, 'YCSB++: Benchmarking and performance debugging advanced features in scalable table stores', in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, ACM, Cascais, Portugal, p. 9.

Patterson, D., Gibson, G. & Katz, R. 1988, 'A case for redundant arrays of inexpensive disks (RAID)', in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ACM, Chicago, Illinois, pp. 109–116.

Patterson, H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S. & Owara, S. 2002, 'SnapMirror®: File system based asynchronous mirroring for disaster recovery', in

Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST), USENIX Association, Monterey, CA, USA, pp. 9–9.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S. & Stonebraker, M. 2009, 'A comparison of approaches to large-scale data analysis', in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ACM, pp. 165–178.

Plaxton, C. G., Rajaraman, R. & Richa, A. W. 1999, 'Accessing nearby copies of replicated objects in a distributed environment', *Theory of Computing Systems*, vol. 32, no. 3, pp. 241–280.

Popek, G. J. & Goldberg, R. P. 1974, 'Formal requirements for virtualizable third generation architectures', *Communications of the ACM*, vol. 17, no. 7, pp. 412–421.

Ramasubramanian, V. & Sirer, E. G. 2004, 'Beehive: O (1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays', in *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 4, USENIX Association, San Francisco, CA, USA, pp. 8–8.

Raphael, J. 2013, 'In Pictures: The worst Cloud outages of 2013', Accessed 20 March 2014, `<http://www.pcworld.idg.com.au/slideshow/466392/pictures_worst_cloud_outages_2013_far_/>`.

Ratnasamy, S., Francis, P., Handley, M., Karp, R. & Shenker, S. 2001, 'A scalable content-addressable network', *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 161–172.

Redis 2009, 'Redis is an open source, BSD licensed, advanced key-value store', Accessed 18 January 2009, `<http://www.redis.io/>`.

Rodrigues, R. & Liskov, B. 2005, 'High availability in DHTs: Erasure coding vs. replication', in *Peer-to-Peer Systems IV*, Springer, pp. 226–239.

Rowstron, A. & Druschel, P. 2001, 'Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems', in *ACM/IFIP/USENIX International Middleware Conference 2001*, Springer, Heidelberg, Germany, pp. 329–350.

Saito, Y., Frølund, S., Veitch, A., Merchant, A. & Spence, S. 2004, 'FAB: Building distributed enterprise disk arrays from commodity components', *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5, pp. 48–58.

Savinov, S. & Daudjee, K. 2010, 'Dynamic database replica provisioning through virtualization', in *Proceedings of the 2nd International workshop on Cloud data management*, ACM, pp. 41–46.

Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I. & Busse, R. 2002, 'XMark: A benchmark for XML data management', in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, VLDB Endowment, pp. 974–985.

Seeger, M. 2009, 'Key-value stores: A practical overview', *Computer Science and Media*. Accessed 07 December 2010, <http://blog.marc-seeger.de/assets/papers/Ultra_Large_Sites_SS09-Seeger_Key_Value_Stores.pdf>.

Seltzer, M. & Bostic, K. 1986, 'Oracle Berkeley DB products', Accessed 03 September 2011, <http://www.oracle.com/us/products/database/berkeley-db/overview/index.html>.

Sheth, A. P. & Larson, J. A. 1990, 'Federated database systems for managing distributed, heterogeneous, and autonomous databases', *ACM Computing Surveys (CSUR)*, vol. 22, no. 3, pp. 183–236.

Shvachko, K., Kuang, H., Radia, S. & Chansler, R. 2010, 'The Hadoop distributed file system', in *26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, IEEE, Incline Village, Nevada, USA, pp. 1–10.

SPC 2013, 'Storage Performance Council: Defining, administrating, and promoting industry-standard, vendor-neutral benchmarks to characterize the performance of storage products', Accessed 22 June 2013, <http://www.storageperformance.org/home/>.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. & Balakrishnan, H. 2001, 'Chord: A scalable peer-to-peer lookup service for internet applications', *ACM SIGCOMM Computer Communication Review*, vol. 31, pp. 149–160.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E. et al. 2005, 'C-store: A column-oriented DBMS', in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, VLDB Endowment, pp. 553–564.

Sullivan III, W. T., Werthimer, D., Bowyer, S., Cobb, J., Gedye, D. & Anderson, D. 1997, 'A new major SETI project based on Project Serendip data and 100,000 personal computers', *Astronomical and Biochemical Origins and the Search for Life in the Universe: Proceedings of the 5th IAU Colloquium International Conference on Bioastronomy*, vol. 1, no. 161, p. 729.

Tamer Özsu, M. & Valduriez, P. 2011, *Principles of Distributed Database Systems*, 3rd ed., Springer.

Tanenbaum, A. S. & Van Steen, M. 2007, *Distributed systems: Principles and paradigms*, 2nd ed., Prentice Hall, Upper Saddle River, NJ.

Tarantool 2009, 'Tarantool: An in-memory NoSQL database', Accessed 12 December 2013, <http://www.tarantool.org/>.

Terracotta Inc. 2009, 'Ehcache: Performance at any scale', Accessed 04 March 2013, <http://www.ehcache.org/>.

Thomas, R. H. 1979, 'A majority consensus approach to concurrency control for multiple copy databases', *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 2, pp. 180–209.

Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P. & Murthy, R. 2009, 'Hive: A warehousing solution over a map-reduce framework', *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 2, pp. 1626–1629.

TPC 2001, 'TPC Benchmarks', Accessed 02 May 2011, <http://www.tpc.org/information/benchmarks.asp>.

Trushkowsky, B., Bodík, P., Fox, A., Franklin, M. J., Jordan, M. I. & Patterson, D. A. 2011, 'The SCADS Director: Scaling a distributed storage system under stringent performance requirements', in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, USENIX Association, San Jose, CA, USA, pp. 163–176.

Van Renesse, R. & Schneider, F. B. 2004, 'Chain replication for supporting high throughput and availability', in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 4, USENIX Association, San Francisco, CA, USA, pp. 91–104.

Van Renesse, R., Minsky, Y. & Hayden, M. 1998, 'A gossip-style failure detection service', in *ACM/IFIP/USENIX International Middleware Conference 1998*, Springer, The Lake District, England, pp. 55–70.

Versant corporation 2008, 'Db4o: Java and .NET Object Database', Accessed 03 September 2011, `<http://www.db4o.com/>`.

Vishwanath, K. V. & Nagappan, N. 2010, 'Characterizing cloud computing hardware reliability', in *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, ACM, Indianapolis, IN, USA, pp. 193–204.

Vo, H. T., Chen, C. & Ooi, B. C. 2010, 'Towards elastic transactional cloud storage with range query support', *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 506–514.

Vogels, W. 2009, 'Eventually consistent', *Communications of the ACM*, vol. 52, no. 1, pp. 40–44.

Voldemort 2009, 'Project Voldemort - A distributed database', Accessed 5 April 2011, `<http://www.project-voldemort.com/voldemort/>`.

Wada, H., Fekete, A., Zhao, L., Lee, K. & Liu, A. 2011, 'Data consistency properties and the trade-offs in commercial cloud storage: The consumers' perspective', in *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, vol. 11, Asilomar, California, USA, pp. 134–143.

Weatherspoon, H. & Kubiatowicz, J. D. 2002, 'Erasure coding vs. replication: A quantitative comparison', in *Peer-to-Peer Systems*, Springer, pp. 328–337.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. & Maltzahn, C. 2006, 'Ceph: A scalable, high-performance distributed file system', in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, Seattle, WA, USA, pp. 307–320.

Wilkinson, K., Sayers, C., Kuno, H. A., Reynolds, D. et al. 2003, 'Efficient RDF Storage and Retrieval in Jena2.', in *Proceedings of the 1st International Workshop on Semantic Web and Databases (SWDB)*, vol. 3, Berlin, Germany, pp. 131–150.

Williams, S. 2000, *The associative model of data*, Lazy Software Ltd., United Kingdom.

You, G. w., Hwang, S. w. & Jain, N. 2011, 'Scalable load balancing in cluster storage systems', in *ACM/IFIP/USENIX International Middleware Conference 2011*, Springer, Lisbon, Portugal, pp. 101–122.

Youseff, L., Butrico, M. & Da Silva, D. 2008, 'Toward a unified ontology of cloud computing', in *Grid Computing Environments Workshop (GCE'08)*, IEEE, pp. 1–10.

Yu, H., Gibbons, P. B. & Nath, S. 2006, 'Availability of multi-object operations', in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, USENIX Association, San Jose, CA, USA, pp. 211–224.

Zaman, S. & Grosu, D. 2011, 'A distributed algorithm for the replica placement problem', *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 22, no. 9, pp. 1455–1468.

Zhang, Q., Cheng, L. & Boutaba, R. 2010, 'Cloud computing: State-of-the-art and research challenges', *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18.

Zhao, B. Y., Kubiatowicz, J. & Joseph, A. D. 2001, 'Tapestry: An infrastructure for fault-tolerant wide-area location and routing', Tech. rep., University of California at Berkeley, CA, USA.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D. & Kubiatowicz, J. D. 2004, 'Tapestry: A resilient global-scale overlay for service deployment', *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 41–53.

Appendix A

Proof of Lemmas

A.1 Proof of Lemma 1

Proof. For all integers $i \in [0, C)$ and $k \in [0, L)$, each $b[i]$ appears at $m_b[x_b][y_b]$ where

$$\begin{cases} x_b = (i + k * C) \mod L \\ y_b = \lfloor \frac{i+k*C}{L} \rfloor \end{cases}.$$

When $C < L$, i.e., $\frac{C}{L} < 1$, there exists at least one violation. For example, let $i = 0$. When $k_1 = 0$ or $k_2 = 1$, $y_b = \lfloor k_1 \frac{C}{L} \rfloor = \lfloor k_2 \frac{C}{L} \rfloor = 0$. That is, $b[0]$ will appear twice at $m_b[0][0]$ and $m_b[C][0]$, which are in the same column.

When $C \geq L$, let $C = L + t$, wherein $t \geq 0$. Thus, for any $k_1 < k_2$, i.e. $k_1 + 1 \leq k_2$, we have $y_b(k_2) > y_b(k_1)$, because:

$$\begin{aligned} y_b(k_2) &= \lfloor \tfrac{i+k_2*C}{L} \rfloor \\ &\geq \lfloor \tfrac{i+(k_1+1)*(L+t)}{L} \rfloor = \lfloor k_1 + \tfrac{i+k_1*t+t}{L} + 1 \rfloor \\ &> \lfloor k_1 + \tfrac{i+k_1*t+t}{L} \rfloor \geq \lfloor k_1 + \tfrac{i+k_1*t}{L} \rfloor \\ &= \lfloor \tfrac{i+k_1*(L+t)}{L} \rfloor \\ &= y_b(k_1). \end{aligned}$$

That is, $y_b(k_1) \neq y_b(k_2)$. Hence, any two replicas of $b[i]$ will not appear in the same column. Similarly, for each $c[i]$, $y_c = C - 1 - \lfloor \frac{i+k*C}{L} \rfloor = C - 1 - y_b$. For any $k_1 \neq k_2$, since $y_b(k_1) \neq y_b(k_2)$, we have $y_c(k_1) \neq y_c(k_2)$. Hence, any two replicas of $c[i]$ will not appear in the same column, either.

□

A.2 Proof of Lemma 2

Proof. Given any two copysets, i.e. $\{a[i_1], b[j_1], c[k_1]\}$ and $\{a[i_2], b[j_2], c[k_2]\}$. We use proof by contradiction. Assume they share more than one node, i.e., $\begin{cases} i_1 = i_2 \\ \\ j_1 = j_2 \end{cases}$, or $\begin{cases} i_1 = i_2 \\ \\ k_1 = k_2 \end{cases}$.

According to the definition of Order 1, $a[i]$ is placed in the i^{th} column. Thus, if $i_1 = i_2$, then $a[i_1]$ and $a[i_2]$ are located in the same column $t = i_1 = i_2$, wherein t is a constant. Since $a[i_1]$ and $b[j_1]$ are in the same copyset, $b[j_1]$ is in column $t_1 = t$, and $b[j_2]$ appears in column $t_2 = t$ as $a[i_2]$ does.

According to Lemma 1, if $j_1 = j_2$, then $b[j_1]$ and $b[j_2]$ appear in two distinct columns, i.e., $t_1 \neq t_2$, which contradicts $t_1 = t_2 = t$. That is to say, $\begin{cases} i_1 = i_2 \\ \\ j_1 = j_2 \end{cases}$ is not valid. Hence, there does not exist two copysets that share the same $a[i]$ and $b[j]$.

Similarly, if $k_1 = k_2$, then $c[k_1]$ and $c[k_2]$ are also located in two distinct columns $t_3 \neq t_4$, which contradict $\{t_3 = t \text{ and } t_4 = t\}$. Hence, there does not exist two copysets that share the same $a[i]$ and $c[k]$, either. □

A.3 Proof of Lemma 3

Proof. Given two integers $i \in [0, C)$ and $k \in [0, L)$.

According to the definition of Order 1, $a[i]$ is placed in the i^{th} column, k^{th} row. Therefore, $a[i]$ appears in each and every row.

According to Order 2, each $b[i]$ appears at $m_b[x_b][y_b]$, wherein $x_b = (i + k * C) \mod L$. We use proof by contradiction. Presume there exist two replicas of $b[i]$, i.e. $k_1 \neq k_2$, such that $x_b = x_b(k_1) = x_b(k_2)$. That is, there exist two integers t_1 and t_2, such that:

$$\begin{cases} i + k_1 * C = t_1 * L + x_b \\ \\ i + k_2 * C = t_2 * L + x_b \end{cases} \tag{A.1}$$

From Equation A.1, we have: $\frac{C}{L} = \frac{t_2 - t_1}{k_2 - k_1}$. Since $\forall k < L$, let $k_1 < k_2$, then $0 < k_2 - k_1 < L$. Since both $(k_2 - k_1)$ and $(t_2 - t_1)$ are integers, there must exist an integer $s > 1$, such that $L = s * (k_2 - k_1)$ and $\frac{C}{L} = \frac{s(t_2 - t_1)}{s(k_2 - k_1)}$. In this case, s is a common divisor of C and L. However, C and L are co-prime, meaning that their greatest common divisor is $s = 1$, which contradicts $s > 1$. Hence, the presumption that there exists $k_1 \neq k_2$ such that

$x_b(k_1) = x_b(k_2)$ is invalid. That is to say, there is no $b[i]$ appearing more than once in any row of the matrix.

Moreover, since each $b[i]$ appears exactly L times in m_b that has exactly L rows, plus there is no row containing more than one $b[i]$, the only possible layout is that each $b[i]$ appears once in each and every row of the matrix.

Similarly, according to Order 3, for each $c[j]$, we have $x_c = (j + k * C) \mod L = x_b$. The conclusion for $b[i]$ is also valid for $c[j]$. □

A.4 Proof of Lemma 4

This proof consists of two parts: i) L is even; and ii) L is odd. In both cases, we prove that, given $0 \leq i, j < C$, when C is the smallest odd number that is greater than L, any combination of $(b[i], c[j])$ appear together in at most one copyset.

A.4.1 Proof for an Even L

Proof. When L is an even number, let $L = 2t$ and $C = 2t + r$, wherein $t \geq 1$ and $r \geq 0$ and both are integers.

According to Order 2, for all integers $k_b \in [0, L)$, each $b[i]$ appears at $m_b[x_b][y_b]$ where:

$$y_b = \lfloor \tfrac{i+k_bC}{L} \rfloor = \lfloor \tfrac{i+k_b*(2t+r)}{2t} \rfloor = \lfloor k_b + \tfrac{i+k_br}{2t} \rfloor$$
$$= k_b + \lfloor \tfrac{i+k_br}{2t} \rfloor$$
$$x_b = (i + k_bC) \mod L$$
$$\implies \quad \lfloor \tfrac{i+k_bC}{L} \rfloor * L + x_b = i + k_bC$$
$$\implies \quad (k_b + \lfloor \tfrac{i+k_br}{2t} \rfloor) * 2t + x_b = i + k_b * (2t + r)$$
$$\implies \quad x_b = i + k_br - 2t \lfloor \tfrac{i+k_br}{2t} \rfloor$$

Similarly, according to Order 3, $\forall k_c \in [0, L)$, each $c[j]$ appears at $m_c[x_c][y_c]$ where:

$$y_c = C - 1 - \lfloor \tfrac{j+k_cC}{L} \rfloor = C - 1 - k_c - \lfloor \tfrac{j+k_cr}{2t} \rfloor$$
$$x_c = (j + k_cC) \mod L$$
$$\implies \quad x_c = j + k_cr - 2t \lfloor \tfrac{j+k_cr}{2t} \rfloor$$

If $(b[i], c[j])$ appears in the same copyset, we have $\begin{cases} x_b = x_c \\ y_b = y_c \end{cases}$. Therefore:

$$\begin{cases} i + k_br - 2t \lfloor \tfrac{i+k_br}{2t} \rfloor = j + k_cr - 2t \lfloor \tfrac{j+k_cr}{2t} \rfloor \\ k_b + \lfloor \tfrac{i+k_br}{2t} \rfloor = C - 1 - k_c - \lfloor \tfrac{j+k_cr}{2t} \rfloor \end{cases} \implies$$

$$\begin{cases} \lfloor \frac{i+k_b r}{2t} \rfloor - \lfloor \frac{j+k_c r}{2t} \rfloor = \frac{i+k_b r}{2t} - \frac{j+k_c r}{2t} \\ \lfloor \frac{i+k_b r}{2t} \rfloor + \lfloor \frac{j+k_c r}{2t} \rfloor = 2t + r - 1 - (k_b + k_c) \end{cases} \tag{A.2}$$

From $\lfloor \frac{i+k_b r}{2t} \rfloor - \lfloor \frac{j+k_c r}{2t} \rfloor = \frac{i+k_b r}{2t} - \frac{j+k_c r}{2t}$, we know $(i + k_b r)$ and $(j + k_c r)$ are congruent modulo $2t$. That is:

$$i + k_b r \equiv j + k_c r \mod 2t \tag{A.3}$$

When $r = 0$, Simultaneous equations A.2 for (k_b, k_c) become:

$$\begin{cases} \lfloor \frac{i}{2t} \rfloor - \lfloor \frac{j}{2t} \rfloor = \frac{i}{2t} - \frac{j}{2t} \\ \lfloor \frac{i}{2t} \rfloor + \lfloor \frac{j}{2t} \rfloor = 2t - 1 - (k_b + k_c) \end{cases}$$, which has only one valid equation for (k_b, k_c).

As a result, for certain $b[i]$ and $c[j]$, there are multiple valid solutions for (k_b, k_c). For example, let $i = j = 0$, then there are $2t$ possible solutions as in Equation A.4. Thus, $r \neq 0$.

$$\begin{cases} k_b = 0 \\ k_c = 2t - 1 \end{cases} \quad \begin{cases} k_b = 1 \\ k_c = 2t - 2 \end{cases} \quad \begin{cases} ..... \\ ..... \end{cases} \quad \begin{cases} k_b = 2t - 1 \\ k_c = 0 \end{cases} \tag{A.4}$$

When $r = 1$, i.e., $C = L + 1$, Simultaneous equations A.2 for (k_b, k_c) become:

$$\begin{cases} \lfloor \frac{i+k_b}{2t} \rfloor - \lfloor \frac{j+k_c}{2t} \rfloor = \frac{i+k_b}{2t} - \frac{j+k_c}{2t} \\ \lfloor \frac{i+k_b}{2t} \rfloor + \lfloor \frac{j+k_c}{2t} \rfloor = 2t - (k_b + k_c) \end{cases} \tag{A.5}$$

We use proof by contradiction to prove that Simultaneous equations A.5 have at most one solution for (k_b, k_c), given any valid value of i, j and t. Assume there are two solutions, i.e. (k_{b1}, k_{c1}) and (k_{b2}, k_{c2}), such that $k_{b1} \neq k_{b2}$ and $k_{c1} \neq k_{c2}$, then we have:

$$\begin{cases} \lfloor \frac{i+k_{b1}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor = \frac{i+k_{b1}}{2t} - \frac{j+k_{c1}}{2t} \\ \lfloor \frac{i+k_{b1}}{2t} \rfloor + \lfloor \frac{j+k_{c1}}{2t} \rfloor = 2t - (k_{b1} + k_{c1}) \\ \lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{j+k_{c2}}{2t} \rfloor = \frac{i+k_{b2}}{2t} - \frac{j+k_{c2}}{2t} \\ \lfloor \frac{i+k_{b2}}{2t} \rfloor + \lfloor \frac{j+k_{c2}}{2t} \rfloor = 2t - (k_{b2} + k_{c2}) \end{cases} \implies$$

$$\begin{aligned} (\lfloor \frac{i + k_{b2}}{2t} \rfloor - \lfloor \frac{i + k_{b1}}{2t} \rfloor) - (\lfloor \frac{j + k_{c2}}{2t} \rfloor - \lfloor \frac{j + k_{c1}}{2t} \rfloor) \\ = \frac{k_{b2} - k_{b1}}{2t} - \frac{k_{c2} - k_{c1}}{2t} \end{aligned} \tag{A.6}$$

$$\begin{aligned} (\lfloor \frac{i + k_{b2}}{2t} \rfloor - \lfloor \frac{i + k_{b1}}{2t} \rfloor) + (\lfloor \frac{j + k_{c2}}{2t} \rfloor - \lfloor \frac{j + k_{c1}}{2t} \rfloor) \\ = -(k_{b2} - k_{b1}) - (k_{c2} - k_{c1}) \end{aligned} \tag{A.7}$$

From Equation A.3, when $r = 1$, we have $\begin{cases} i + k_{b1} \equiv j + k_{c1} \mod 2t \\ i + k_{b2} \equiv j + k_{c2} \mod 2t \end{cases}$

$\implies \quad k_{b2} - k_{b1} \equiv k_{c2} - k_{c1} \mod 2t$

$\implies \quad (k_{b2} - k_{b1}) = (k_{c2} - k_{c1}) \pm n * 2t$, where $n \geq 0$.

Let $k_{b1} < k_{b2}$. Since $\forall k \in [0, L) = [0, 2t)$, then we have $0 < k_{b2} - k_{b1} < 2t$ and $-2t < k_{c2} - k_{c1} < 2t$. Thus, we have Equation A.8, wherein $n \in \{0, 1\}$.

$$(k_{b2} - k_{b1}) = (k_{c2} - k_{c1}) + n * 2t \tag{A.8}$$

When $n = 0$, i.e. $k_{b2} - k_{b1} = k_{c2} - k_{c1} > 0$, then $k_{c1} < k_{c2}$. However, the following equation contradicts Equation A.7. Thus, $n \neq 0$.

$$\begin{cases} (\lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor) + (\lfloor \frac{j+k_{c2}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor) \\ \qquad \geq (\lfloor \frac{i+k_{b1}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor) + (\lfloor \frac{j+k_{c1}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor) = 0 \quad \cdot \\ -(k_{b2} - k_{b1}) - (k_{c2} - k_{c1}) < 0 \end{cases}$$

When $n = 1$ (and $k_{b1} < k_{b2}$), we have $\begin{cases} -2t < k_{c2} - k_{c1} < 0 \\ (k_{b2} - k_{b1}) = (k_{c2} - k_{c1}) + 2t \end{cases}$

$\implies \quad \frac{k_{b2}-k_{b1}}{2t} - \frac{k_{c2}-k_{c1}}{2t} = 1.$

From Equation A.6:

$1 = \frac{k_{b2}-k_{b1}}{2t} - \frac{k_{c2}-k_{c1}}{2t} = (\lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor) - (\lfloor \frac{j+k_{c2}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor)$

Moreover, since $k_b < L$, let $k_b = L - d = 2t - d$, wherein $d > 0$. Since $i \leq C - 1 = 2t$, we have: $\lfloor \frac{i+k_b}{2t} \rfloor \leq \lfloor \frac{2t+2t-d}{2t} \rfloor = 2 + \lfloor \frac{-d}{2t} \rfloor < 2$. Thus, $\lfloor \frac{i+k_b}{2t} \rfloor \in \{0, 1\}$. Similarly, $\lfloor \frac{j+k_c}{2t} \rfloor \in \{0, 1\}$.

Therefore, $\begin{cases} \lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor \in \{-1, 0, 1\} \\ \lfloor \frac{j+k_{c2}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor \in \{-1, 0, 1\} \\ (\lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor) - (\lfloor \frac{j+k_{c2}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor) = 1 \end{cases}$

$\implies \quad \begin{cases} \lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor = 0 \\ \lfloor \frac{j+k_{c2}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor = -1 \end{cases}$, or $\begin{cases} \lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor = 1 \\ \lfloor \frac{j+k_{c2}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor = 0 \end{cases}$

That is, $(\lfloor \frac{i+k_{b2}}{2t} \rfloor - \lfloor \frac{i+k_{b1}}{2t} \rfloor) + (\lfloor \frac{j+k_{c2}}{2t} \rfloor - \lfloor \frac{j+k_{c1}}{2t} \rfloor) = \pm 1$.

Simultaneous equations A.6 and A.7 become $\begin{cases} (k_{b2} - k_{b1}) - (k_{c2} - k_{c1}) = 2t \\ -(k_{b2} - k_{b1}) - (k_{c2} - k_{c1}) = \pm 1 \end{cases}$

$\implies \quad \begin{cases} k_{b2} - k_{b1} = t \mp 1/2 \\ k_{c2} - k_{c1} = -t \pm 1/2 \end{cases}$

However, since $\forall k$ are integers, $(k_{b2} - k_{b1})$ or $(k_{c2} - k_{c1})$ cannot be non-integers. That is to say, when $r = 1$, there does not exist two distinct solutions (k_{b1}, k_{c1}) and (k_{b2}, k_{c2}) for any two nodes $b[i]$ and $c[j]$. Hence, when L is an even number and $C = L + 1$, any combination of $(b[i], c[j])$ appear together in at most one copyset.

□

A.4.2 Proof for an Odd L

Proof. When L is an odd number, let $L = 2t - 1$ and $C = 2t + r$, wherein $t \geq 1$ and $r \geq -1$ and both are integers.

Similar to the proof in Appendix A.4.1, for all integers $k_b \in [0, L)$, each $b[i]$ appears at $m_b[x_b][y_b]$ where:

$$y_b = \lfloor \tfrac{i+k_bC}{L} \rfloor = \lfloor \tfrac{i+k_b*(2t+r)}{2t-1} \rfloor = k_b + \lfloor \tfrac{i+k_b*(r+1)}{2t-1} \rfloor$$

$$x_b = (i + k_bC) \mod L$$

$$\implies \quad \lfloor \tfrac{i+k_bC}{L} \rfloor * L + x_b = i + k_bC$$

$$\implies \quad (k_b + \lfloor \tfrac{i+k_b*(r+1)}{2t-1} \rfloor) * (2t-1) + x_b = i + k_b * (2t + r)$$

$$\implies \quad x_b = i + k_b * (r+1) - (2t-1)\lfloor \tfrac{i+k_b*(r+1)}{2t-1} \rfloor$$

Similarly, for all integers $k_c \in [0, L)$, each $c[j]$ appears at $m_c[x][y]$ where:

$$y_c = C - 1 - \lfloor \tfrac{j+k_cC}{L} \rfloor = C - 1 - k_c - \lfloor \tfrac{j+k_c*(r+1)}{2t-1} \rfloor$$

$$x_c = (j + k_cC) \mod L$$

$$\implies \quad x_c = j + k_c * (r+1) - (2t-1)\lfloor \tfrac{j+k_c*(r+1)}{2t-1} \rfloor$$

If $(b[i], c[j])$ appears in the same copyset, we have $\begin{cases} x_b = x_c \\ y_b = y_c \end{cases}$. Hence,

$$\begin{cases} i + k_b * (r+1) - (2t-1)\lfloor \tfrac{i+k_b*(r+1)}{2t-1} \rfloor = j + k_c * (r+1) - (2t-1)\lfloor \tfrac{j+k_c*(r+1)}{2t-1} \rfloor \\ k_b + \lfloor \tfrac{i+k_b*(r+1)}{2t-1} \rfloor = C - 1 - k_c - \lfloor \tfrac{j+k_c*(r+1)}{2t-1} \rfloor \end{cases} \implies$$

$$\begin{cases} \lfloor \tfrac{i+k_b*(r+1)}{2t-1} \rfloor - \lfloor \tfrac{j+k_c*(r+1)}{2t-1} \rfloor = \tfrac{i+k_b*(r+1)}{2t-1} - \tfrac{j+k_c*(r+1)}{2t-1} \\ \lfloor \tfrac{i+k_b*(r+1)}{2t-1} \rfloor + \lfloor \tfrac{j+k_c*(r+1)}{2t-1} \rfloor = 2t + r - 1 - (k_b + k_c) \end{cases} \tag{A.9}$$

When $r = -1$, i.e., $C = L$, Simultaneous equations A.9 for (k_b, k_c) become

$\begin{cases} \lfloor \tfrac{i}{2t-1} \rfloor - \lfloor \tfrac{j}{2t-1} \rfloor = \tfrac{i}{2t-1} - \tfrac{j}{2t-1} \\ \lfloor \tfrac{i}{2t-1} \rfloor + \lfloor \tfrac{j}{2t-1} \rfloor = 2t - 2 - (k_b + k_c) \end{cases}$, which has only one valid equation for (k_b, k_c).

As a result, for certain $b[i]$ and $c[j]$, there are multiple valid solutions for (k_b, k_c). For ex-

ample, let $i = j = 0$, then there are $2t - 1$ possible solutions as in Equation A.10. Thus, $r \neq -1$.

$$\begin{cases} k_b = 0 \\ k_c = 2t-2 \end{cases} \quad \begin{cases} k_b = 1 \\ k_c = 2t-3 \end{cases} \quad \begin{cases} ..... \\ ..... \end{cases} \quad \begin{cases} k_b = 2t-2 \\ k_c = 0 \end{cases} \tag{A.10}$$

When $r = 0$, i.e., $C = L + 1$, Simultaneous equations A.9 for (k_b, k_c) become:

$$\begin{cases} \lfloor \frac{i+k_b}{2t-1} \rfloor - \lfloor \frac{j+k_c}{2t-1} \rfloor = \frac{i+k_b}{2t-1} - \frac{j+k_c}{2t-1} \\ \lfloor \frac{i+k_b}{2t-1} \rfloor + \lfloor \frac{j+k_c}{2t-1} \rfloor = 2t - 1 - (k_b + k_c) \end{cases} \tag{A.11}$$

There exists at least two distinct solutions of (k_b, k_c) for certain combinations of (i, j). For example, when $i = 0, j = 1$, Equation A.11 has two distinct solutions as shown in Equation A.12. Thus, $r \neq 0$.

$$\begin{cases} k_b = 0 \\ k_c = 2t-2 \end{cases} \quad \begin{cases} k_b = t \\ k_c = t-1 \end{cases} \tag{A.12}$$

When $r = 1$, i.e., $C = L + 2$, Simultaneous equations A.9 for (k_b, k_c) become:

$$\begin{cases} \lfloor \frac{i+2k_b}{2t-1} \rfloor - \lfloor \frac{j+2k_c}{2t-1} \rfloor = \frac{i+2k_b}{2t-1} - \frac{j+2k_c}{2t-1} \\ \lfloor \frac{i+2k_b}{2t-1} \rfloor + \lfloor \frac{j+2k_c}{2t-1} \rfloor = 2t - (k_b + k_c) \end{cases} \tag{A.13}$$

Similarly, we use proof by contradiction to prove that Simultaneous equations A.13 have at most one solution for (k_b, k_c), given any valid value of i, j and t. Assume there are two solutions, i.e. (k_{b1}, k_{c1}) and (k_{b2}, k_{c2}), such that $k_{b1} \neq k_{b2}$ and $k_{c1} \neq k_{c2}$, then we have:

$$\begin{aligned} (\lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor) - (\lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor) \\ = \frac{2(k_{b2}-k_{b1})}{2t-1} - \frac{2(k_{c2}-k_{c1})}{2t-1} \end{aligned} \tag{A.14}$$

$$\begin{aligned} (\lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor) + (\lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor) \\ = -(k_{b2}-k_{b1}) - (k_{c2}-k_{c1}) \end{aligned} \tag{A.15}$$

Similarly, when $r = 1$, we have $\begin{cases} i + 2k_{b1} \equiv j + 2k_{c1} \mod (2t-1) \\ i + 2k_{b2} \equiv j + 2k_{c2} \mod (2t-1) \end{cases}$

$\implies \quad 2(k_{b2} - k_{b1}) \equiv 2(k_{c2} - k_{c1}) \mod (2t-1)$

$\implies \quad (2(k_{b2} - k_{b1}) = 2(k_{c2} - k_{c1}) \pm n * (2t-1)$, where $n \geq 0$.

Let $k_{b1} < k_{b2}$. Since $\forall k \in [0, 2t-1)$, thus $\begin{cases} 0 < 2(k_{b2} - k_{b1}) < 2(2t-1) \\ -2(2t-1) < 2(k_{c2} - k_{c1}) < 2(2t-1) \end{cases}$.
Moreover, since both $2(k_{b2} - k_{b1})$ and $2(k_{c2} - k_{c1})$ are even numbers, while $(2t-1)$ is odd, Then n must be even. Therefore, we have Equation A.16, wherein $n \in \{0, 2\}$.

$$2(k_{b2} - k_{b1}) = 2(k_{c2} - k_{c1}) + n * (2t-1) \tag{A.16}$$

When $n = 0$, i.e. $k_{b2} - k_{b1} = k_{c2} - k_{c1} > 0$, then $k_{c1} < k_{c2}$. However, the following equation contradicts Equation A.15. Thus, $n \neq 0$.

$$\begin{cases} (\lfloor \frac{i+k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+k_{b1}}{2t-1} \rfloor) + (\lfloor \frac{j+k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+k_{c1}}{2t-1} \rfloor) \geq 0 \\ -(k_{b2} - k_{b1}) - (k_{c2} - k_{c1}) < 0 \end{cases}.$$

When $n = 2$, we have $\begin{cases} -(2t-1) < k_{c2} - k_{c1} < 0 \\ (k_{b2} - k_{b1}) = (k_{c2} - k_{c1}) + (2t-1) \end{cases}$

$\implies \quad \frac{k_{b2}-k_{b1}}{2t-1} - \frac{k_{c2}-k_{c1}}{2t-1} = 1.$

From Equation A.14:

$(\lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor) - (\lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor) = 2(\frac{k_{b2}-k_{b1}}{2t-1} - \frac{k_{c2}-k_{c1}}{2t-1}) = 2.$

Moreover, since $k_b < L$, let $k_b = L - s = 2t - 1 - s$, wherein $s \geq 1$. Since $i \leq C - 1 = 2t$, we have: $\lfloor \frac{i+2k_b}{2t-1} \rfloor \leq \lfloor \frac{2t+4t-2-2s}{2t-1} \rfloor = 3 + \lfloor -\frac{2s-1}{2t-1} \rfloor < 3$. Thus, $\lfloor \frac{i+2k_b}{2t-1} \rfloor \in \{0, 1, 2\}$. Similarly, $\lfloor \frac{j+2k_c}{2t-1} \rfloor \in \{0, 1, 2\}$. Therefore, $\begin{cases} \lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor \in \{-2, -1, 0, 1, 2\} \\ \lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor \in \{-2, -1, 0, 1, 2\} \\ (\lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor) - (\lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor) = 2 \\ k_{b2} > k_{b1}, k_{c2} < k_{c1} \end{cases}$

$$\implies \quad \begin{cases} \lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor = 0 \\ \lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor = -2 \end{cases}, \text{ or } \begin{cases} \lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor = 1 \\ \lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor = -1 \end{cases}, \text{ or}$$

$$\begin{cases} \lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor = 2 \\ \lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor = 0 \end{cases}.$$

That is, $(\lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor) + (\lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor) \in \{-2, 0, 2\}$. Let $n \in \{-1, 0, 1\}$. Therefore, $(\lfloor \frac{i+2k_{b2}}{2t-1} \rfloor - \lfloor \frac{i+2k_{b1}}{2t-1} \rfloor) + (\lfloor \frac{j+2k_{c2}}{2t-1} \rfloor - \lfloor \frac{j+2k_{c1}}{2t-1} \rfloor) = 2n$.

Simultaneous equations A.14 and A.15 become $\begin{cases} (k_{b2} - k_{b1}) - (k_{c2} - k_{c1}) = 2t - 1 \\ -(k_{b2} - k_{b1}) - (k_{c2} - k_{c1}) = 2n \end{cases}$

$$\Longrightarrow \quad \begin{cases} k_{b2} - k_{b1} = t - n - 1/2 \\ k_{c2} - k_{c1} = -t - n + 1/2 \end{cases}$$

However, since $\forall k$ are integers, $(k_{b2} - k_{b1})$ or $(k_{c2} - k_{c1})$ cannot be non-integers. That is to say, when $r = 1$, there does not exist two distinct solutions (k_{b1}, k_{c1}) and (k_{b2}, k_{c2}) for any two nodes $b[i]$ and $c[j]$. Hence, when L is an odd number and $C = L + 2$, any combination of $(b[i], c[j])$ appear together in at most one copyset.

□

A.5 Proof of Lemma 5

Proof. Given any two copysets, i.e. $(a[i_1], b[j_1], c[k_1])$ and $(a[i_2], b[j_2], c[k_2])$. According to Lemma 2, all the possible solutions are:

$$\begin{cases} i_1 \neq i_2 \\ j_1 \neq j_2 \\ k_1 \neq k_2 \end{cases}, \text{ or } \begin{cases} i_1 = i_2 \\ j_1 \neq j_2 \\ k_1 \neq k_2 \end{cases}, \text{ or } \begin{cases} i_1 \neq i_2 \\ j_1 = j_2 \end{cases}, \text{ or } \begin{cases} i_1 \neq i_2 \\ k_1 = k_2 \end{cases}.$$

Moreover, according to Lemma 4, if $j_1 = j_2$, then $k_1 \neq k_2$. Conversely, if $k_1 = k_2$, then $j_1 \neq j_2$. Therefore, the solutions presented above become:

$$\begin{cases} i_1 \neq i_2 \\ j_1 \neq j_2 \\ k_1 \neq k_2 \end{cases}, \text{ or } \begin{cases} i_1 = i_2 \\ j_1 \neq j_2 \\ k_1 \neq k_2 \end{cases}, \text{ or } \begin{cases} i_1 \neq i_2 \\ j_1 = j_2 \\ k_1 \neq k_2 \end{cases}, \text{ or } \begin{cases} i_1 \neq i_2 \\ j_1 \neq j_2 \\ k_1 = k_2 \end{cases}.$$

As can be seen, in any one solution, there is at most one common node. That is to say, any two copysets share at most one common node.

□

A.6 Proof of Lemma 6

Proof. Since C is the smallest odd number greater than L, for all integers $t > 0$, we have $\begin{cases} L = 2t - 1 \\ C = 2t + 1 \end{cases}$, or $\begin{cases} L = 2t \\ C = 2t + 1 \end{cases}$. In any case, L and C are co-prime. According to Lemma 3, every node appears once in every row of the matrix, in which there are L rows. Since nodes from different rows cannot form a copyset, thus each node belongs to L distinct copysets.

For each copyset that the node n_i belongs to, there are $R-1$ other distinct nodes. Moreover, according to Lemma 5, any two of these copysets share at most one common node. Hence, the shared node can only be n_i. Therefore, these copysets contain $L*(R-1)$ distinct nodes other than n_i. Since $S = L*(R-1)$, thus each node shares copysets with S other distinct nodes.

□