UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF SCIENCES AND ENGINEERING

Efficient adaptation mechanisms for improving performance during internal or external changes in distributed data stores

Antonios Papaioannou

PhD Dissertation

Submitted

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, November 2021

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

Efficient adaptation mechanisms for improving performance during internal or external changes in distributed data stores

PhD Dissertation Presented

by **Antonios Papaioannou**

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Antonios Papaioannou

Supervisor: Kostas Magoutis, Associate Professor, University of Crete

---

Committee Member: Dimitris Plexousakis, Professor, University of Crete

---

Committee Member: Evangelos Markatos, Professor, University of Crete

---

Committee Member: Angelos Bilas, Professor, University of Crete

---

Committee Member: Polyvios Pratikakis, Assistant Professor, University of Crete

---

Committee Member: Nikos Parlavantzas, Associate Professor, Institut National des Sciences Appliquées de Rennes

---

Committee Member: Evangelia Kalyvianaki, Senior Lecturer, University of Cambridge

---

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, November 2021

Abstract

The evolution of distributed data management systems, especially a class of systems developed during the last 15-20 years commonly known as NoSQL data stores, has led to a multitude of designs optimised for different application types, data formats, and workload characteristics. Given the complexity of the environments they operate in as parts of multi-tier software stacks driven by Internet workloads, data stores are facing significant challenges during their operation. An important objective for service operators is to ensure that data store performance levels and guarantees are maintained despite internal or external changes that they face. Such an objective can be reached via automated adaptation mechanisms by which data stores adapt to changes automatically and transparently while maintaining efficiency and performance goals as the data store transitions to new configurations.

In this dissertation we explore adaptation mechanisms in distributed data stores facing internally or externally-induced changes, with a focus on workload variations, occasional background activities, or the evolution of an external middleware component that interoperates with a distributed data store. We propose novel adaptation mechanisms and improvements to existing mechanisms in three different contexts (data store elasticity, masking background activities, and alignment with external distributed middleware), aiming to improve the overall performance during the aforementioned contexts in the lifecycle of scalable data stores, aiming at challenges that had not been addressed so far.

First, this dissertation focuses on the expansion phase of a data store when the need arises to adapt its capacity as workload demands increase and the system tries to improve its performance by incorporating more resources. We study the performance impact of data transfers over the network during this phase and propose a mechanism that schedules data transfers in a fine-grain manner, reducing their performance impact while progressively increasing the processing capacity in an incremental fashion. The proposed

method realizes early benefits from data transfers during the elasticity action as it incorporates new resources and makes data sub-sections available prior to completing the full data transfers.

Next, we study the performance overhead of background activities that often impact data store performance. We propose replica-group reconfiguration as a way to mask performance bottlenecks in replicated data stores and investigate the benefits of changing replica-group leadership prior to resource-intensive background tasks (e.g. internal data reorganization, garbage collection or data backup tasks). Our observation of an occasional performance glitch during reconfiguration actions, caused by cold-cache misses in the cache of a new leader that was not adequately prepared for the transition to the new configuration, led us to propose a new mechanism to maintain up-to-date read caches across replicas without affecting the data consistency and availability by disseminating read-hints within the replica group.

Finally, in this dissertation we investigate the benefits of automatically aligning data stores with distributed middleware systems that rely on those data stores to maintain their state. We do that by appropriately co-locating data partitions of data store with processing tasks of the distributed middleware systems. We propose a system that continuously strives to discover such alignment opportunities across systems and improve data locality. The alignment actions combine multiple data store mechanisms in common use, such as data replication and migration, as well as the adaptation of the partitioning schemes across systems, a mechanism that has not been studied before in this context.

The evaluation of the proposed mechanisms over widely deployed systems confirms their performance improvements, advancing the state of the art in distributed data stores in the direction of systems that adapt more efficiently and in new ways through internal and external changes in their lifecycle.

Keywords: distributed data stores, adaptation, performance management

Supervisor: Kostas Magoutis
Associate Professor
Computer Science Department
University of Crete

Περίληψη

Η εξέλιξη στον χώρο των κατανεμημένων συστημάτων διαχείρισης δεδομένων, ιδιαίτερα μιας κατηγορίας τέτοιων συστημάτων που αναπτύχθηκαν τα τελευταία 15-20 χρόνια γνωστά και ως συστήματα **NoSQL**, είναι ραγδαία. Μια πληθώρα τέτοιων συστημάτων βασίζεται σε ποικίλες σχεδιαστικές αποφάσεις που εστιάζουν στην βελτιστοποίηση για συγκεκριμένους τύπους εφαρμογών, μορφών δεδομένων και χαρακτηριστικών φόρτου εργασίας. Τα συστήματα αυτά αποτελούν συνήθως μέρος πολυεπίπεδων εφαρμογών ικανών να εξυπηρετούν φόρτο μεγάλης κλίμακας (συνήθως προερχόμενο από το **Internet**) κάτω από πολύπλοκες συνθήκες. Ένας σημαντικός στόχος για τους παρόχους υπηρεσιών είναι να διασφαλίσουν το επίπεδο απόδοσης των συστημάτων αυτών παρά τις εσωτερικές ή εξωτερικές αλλαγές και προκλήσεις που αντιμετωπίζουν. Ένας τέτοιος στόχος μπορεί να επιτευχθεί μέσω μηχανισμών προσαρμογής που επιτρέπουν στα συστήματα διαχείρισης δεδομένων να προσαρμόζονται στις αλλαγές αυτόματα, διατηρώντας παράλληλα τους στόχους και την αποδοτικότητα τους.

Σε αυτή τη διατριβή διερευνούμε μηχανισμούς προσαρμογής σε κατανεμημένα συστήματα διαχείρισης δεδομένων που έρχονται αντιμέτωπα με αλλαγές που πηγάζουν είτε από το εσωτερικό του ίδιου του συστήματος ή από εξωγενείς παράγοντες. Μελετάμε ιδιαίτερα αλλαγές που αντιμετωπίζει το σύστημα λόγω αύξησης του φόρτου εργασίας, την εκτέλεση έκτακτων/περιοδικών δραστηριοτήτων στο παρασκήνιο ή την αλληλεπίδραση και συνέργεια με εξωτερικά συστήματα τα οποία εξελίσσονται ανεξάρτητα και παράλληλα με τα συστήματα διαχείρισης δεδομένων. Προτείνουμε νέους μηχανισμούς προσαρμογής των συστημάτων καθώς και βελτιώσεις σε υφιστάμενους μηχανισμούς σε τρία διαφορετικά πλαίσια (ελαστικότητα του συστήματος, διαχείριση της επίπτωσης δραστηριοτήτων που εκτελούνται στο παρασκήνιο και αποδοτικότερη συνέργεια και αλληλεπίδραση με εξωτερικά συστήματα επεξεργασίας δεδομένων). Μελετάμε προκλήσεις που δεν είχαν αντιμετωπιστεί μέχρι σήμερα με στόχο τη βελτίωση της συνολικής απόδοσης των συστημάτων.

Αρχικά εστιάζουμε στην φάση της επέκτασης του συστήματος κατά τη διάρκεια της οποίας

ενσωματώνονται νέοι πόροι ώστε να βελτιωθεί η απόδοσή του και να ανταποκριθεί στην αύξηση του φόρτου και των προκλήσεων που αυτή συνεπάγεται. Τα δεδομένα ανακατανέμονται εσωτερικά ώστε οι νέοι πόροι να λάβουν το αναλογούν τους μερίδιο φόρτου. Μελετάμε τον αντίκτυπο στην απόδοση του συστήματος εξαιτίας της μεταφοράς δεδομένων πάνω από το δίκτυο κατά αυτή την ανακατανομή των δεδομένων. Προτείνουμε έναν μηχανισμό ο οποίος δημιουργεί και εφαρμόζει ένα πλάνο μεταφοράς των δεδομένων σε μικρότερα αυτόνομα τμήματα. Κάθε φορά που η μεταφορά ενός τμήματος των δεδομένων ολοκληρώνεται, η εξυπηρέτηση του φόρτου που αντιστοιχεί στα δεδομένα αυτά γίνεται από τους νέους πόρους. Με αυτόν τον τρόπο το συνολικό σύστημα ενεργοποιεί οφέλη των νέων πόρων καθώς η διαδικασία επέκτασης βρίσκεται ακόμα υπό εξέλιξη (πριν ολοκληρωθεί η πλήρης μεταφορά όλων των δεδομένων) με αποτέλεσμα να αυξάνεται προοδευτικά η συνολική χωρητικότητα του συστήματος.

Στη συνέχεια, μελετάμε δραστηριότητες που εκτελούνται στο παρασκήνιο (λειτουργίες για την εσωτερική αναδιοργάνωση των δεδομένων ή ενέργειες δημιουργίας αντιγράφων ασφάλειας) οι οποίες καταναλώνουν πόρους του συστήματος με αποτέλεσμα να έχουν αρνητική επίδραση στην απόδοση του συστήματος. Οι κόμβοι των συστημάτων αυτών είναι συνήθως οργανωμένοι σε ομάδες αντιγράφων για λόγους διαθεσιμότητας των δεδομένων. Συστήματα τα οποία υποστηρίζουν ισχυρή συνέπεια των δεδομένων περιορίζουν την ανάγνωση και εγγραφή των δεδομένων σε ένα υποσύνολο κόμβων. Προτείνουμε ένα μηχανισμό προσαρμογής της ομάδας αντιγράφων ώστε οι κόμβοι που εξυπηρετούν κυρίως τον φόρτο των χρηστών να είναι αυτοί οι οποίοι δεν έχουν ενεργές διεργασίες παρασκηνίου. Επιπλέον παρατηρούμε οτι ο μηχανισμός αλλαγής των κόμβων που εξυπηρετούν τον φόρτο που παράγουν οι χρήστες εμφανίζει μια αδυναμία η οποία σχετίζεται με την κρυφή μνήμη (cache). Αυτή οφειλεται στο ότι τα δευτερεύοντα αντίγραφα δεν ενημερώνουν πλήρως την cache τους με αποτέλεσμα να μην είναι επαρκώς προετοιμασμένα όταν καλούνται να ξεκινήσουν να εξυπηρετούν τους χρήστες. Αυτή η παρατήρηση μας οδήγησε να προτείνουμε έναν νέο μηχανισμό που επιτρέπει σε δευτερεύοντα αντίγραφα να διατηρούν ενημερωμένες caches. Ο μηχανισμός που προτείνουμε βασίζεται στην διάδοση των εντολών ανάγνωσης σε όλους τους κόμβους που φιλοξενούν αντίγραφα των δεδομένων χωρίς να αλλοιώνεται το μοντέλο συνέπειας και διαθεσιμότητας των δεδομένων του συστήματος.

Τέλος, στη διατριβή αυτή διερευνούμε τα οφέλη του αυτόματου συντονισμού κατανεμημένων συστημάτων διαχείρισης δεδομένων με εξωτερικά συστήματα με τα οποία αυτά αλληλεπιδρούν (π.χ. κατανεμημένα συστήματα επεξεργασίας δεδομένων) μέσω του συντονισμού της ανάπτυξης των συστημάτων επί των υπολογιστικών και αποθηκευτικών πόρων. Προτείνουμε ένα σύστημα που διερευνά συνεχώς ευκαιρίες συντονισμού μεταξύ των συστημάτων και βελτιώνει την τοποθεσία των δεδομένων ώστε να βρίσκονται εγγύτερα στους πόρους επεξεργασίας τους. Ο συντονισμός των συστημάτων επιτυγχάνεται χρησιμοποιώντας μηχανισμούς προσαρμογής που σχετίζονται με την δημιουργία αντιγράφων και την μεταφορά δεδομένων. Επιπλέον μελετάμε

για πρώτη φορά σε αυτό το πλαίσιο την προσαρμογή του μηχανισμού με τον οποίο τα δεδομένα διαμερίζονται και κατανέμονται στους κόμβους του συστήματος.

Η πειραματική αξιολόγηση των μηχανισμών που μελετώνται σε αυτή την διατριβή επιβεβαιώνει τα οφέλη τους στην βελτίωση της απόδοσης ευρέως διαδεδομένων συστημάτων. Η μελέτη αυτή συμβάλλει στην εξέλιξη των συστημάτων διαχείρισης δεδομένων προς την κατεύθυνση συστημάτων που μπορούν να προσαρμόζονται αποδοτικά καθώς έρχονται αντιμέτωπα με εσωτερικές ή εξωτερικές αλλαγές κατά την διάρκεια του κύκλου ζωής τους.

Λέξεις κλειδία: Κατανεμημένα συστήματα διαχείρισης δεδομένων, προσαρμογή συστημάτων, διαχείριση απόδοσης συστημάτων

Επόπτης: Κώστας Μαγκούτης
Αναπληρωτής Καθηγητής
Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Acknowledgments

There are many people who helped me along the way on this journey; people who significantly influenced and inspired my life during my graduate studies.

First and foremost I would like to express my sincere gratitude to my advisor Prof. Kostas Magoutis for his unwavering support and belief in me. Through our long discussions and debates, I gained invaluable knowledge and learned how to scientifically address research problems. He has been an endless source of inspiration, guidance, support and encouragement. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank my advisory committee members: Prof. Dimitris Plexousakis, and Prof. Evangelos Markatos for their insightful comments, suggestions and feedback which incented me to widen my research from various perspectives. I would like to extend my sincere thanks to the examination committee members: Prof. Angelos Bilas, Prof. Polyvios Pratikakis, Prof. Nikos Parlavantzas and Prof. Evangelia Kalyvianaki for their insightful comments and suggestions helping me to prepare and improve the final version of the thesis.

I would like to thank the Hellenic Foundation for Research & Innovation (H.F.R.I.) for partly supporting this dissertation through an individual scholarship for PhD Candidates (2017-2018). I am grateful to the Institute of Computer Science (ICS) of the Foundation for Research and Technology-Hellas (FORTH) for giving me the opportunity to work and collaborate with brilliant researchers on interesting projects and also supporting me with graduate research scholarships throughout my doctoral studies. Funding for my dissertation came from several national and European research projects. I thankfully acknowledge

the H.F.R.I. STREAMSTORE faculty grant (Grant ID HFRI-FM17-1998), the EVOLVE H2020 (GA no. 825061), and the PaaSage FP7 (GA no. FP7-317715) projects, for funding my research. I am also honored to be awarded the Maria Michael Manasaki legacy's fellowship for the academic year 2019-2020.

Working on interesting research ideas is hard and challenging but also fun when you are surrounded by beautiful people. Many thanks to Giorgos Vasiliadis, Michalis Giaourtas, Christos Papachristos, Antonis Krithinakis, Manos Papoutsakis, Eva Papadogiannaki, Manos Pavlidakis, Iakovos Kolokasis and Maria Oikonomidou for having fun times in and outside of the lab.

I consider myself very fortunate for enjoying the friendship of wonderful people during my time in Heraklion. I would like to express my sincere appreciation to Tasos Papagiannis, Vassilis Papakonstantinou, Ilias Batzelios, Serafeim Chrisovergis, Maria Chalkiadaki and Ria Vrouva for the love, encouragement and the great moments we had together all these years.

Last but not least, I would like to express my gratitude to my parents, Maria and Alexandros, and my brother Giorgos. Without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete my study. My appreciation also goes out to my aunt Despina and my uncle Giorgos for their support during my time in Heraklion.

Antonis Papaioannou
Heraklion, November 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The profound need for scalable data storage

In recent years, the production and storage of data at a global scale has grown nearly exponentially and to date there is no sign of this trend slowing down. The key drivers for the rise of *Big Data*, a domain of technology referring to the pervasive growth in the production, storage, and processing of vast quantities of data, is mainly driven by successful Internet-scale applications processing social media, text messages, and media files, and by the Internet of Things (IoT) where billions of connected devices continuously generate new data. At a global scale, data is collected from an increased number of sources and is consumed by an increasing number of processing tasks. IDC forecasts the global data space will reach 175 zettabytes[1] by 2025 [161]. Cloud computing [54] and other micro-architectural trends [167, 193] further promote global data access, allowing Internet end-users to do most of their activities online, from business communications to shopping and social networking. It is expected that by 2025, 49% of the world's stored data will reside in public cloud environments [161]. The profound need for efficient and scalable data management systems in such infrastructures is today more critical than ever before.

Especially during the last decade, commodity data management systems have evolved

[1] A zettabyte is a trillion gigabytes. As it is hard to imagine the scale of 175 zettabytes, we quote the example described by David Reinsel, senior vice president at IDC: *"If one were able to store 175ZB onto BluRay discs, then you'd have a stack of discs that can get you to the moon 23 times"*

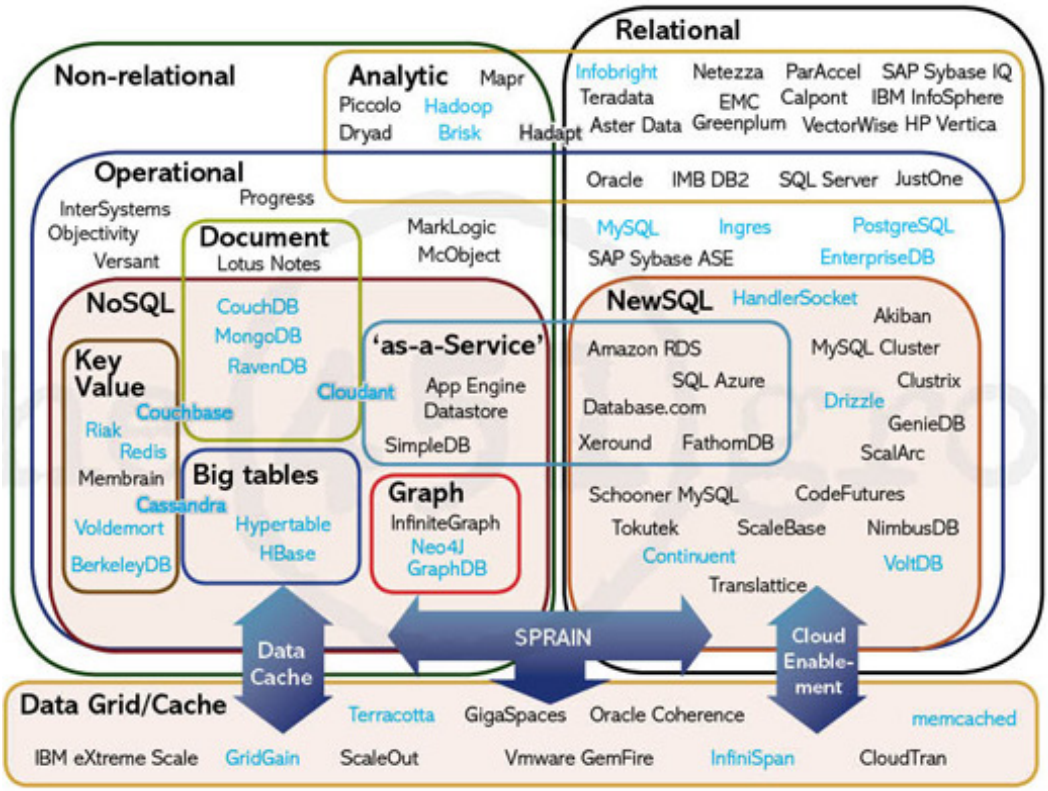


Figure 1.1: A classification of data management systems available commercially or in open-source form today [29]

considerably along several directions to accommodate the evolving demand for storing and managing large amounts of data. Data are diverse, coming in multiple formats; applications use different data structures with specific data access patterns and have different consistency, durability and availability needs, requiring specific query-processing architectures and programming interfaces. The emerging application requirements lead to a plethora of data management systems today optimised for different application types, data formats and workload characteristics (Figure 1.1). Application designers often face a dilemma when choosing an appropriate system for their requirements. Multiple factors, such as the workload characteristics, data access pattern, and the available resources affect the performance of data storage and management systems (also referred to as *data stores* for brevity in this thesis). The decision on which data management system to use

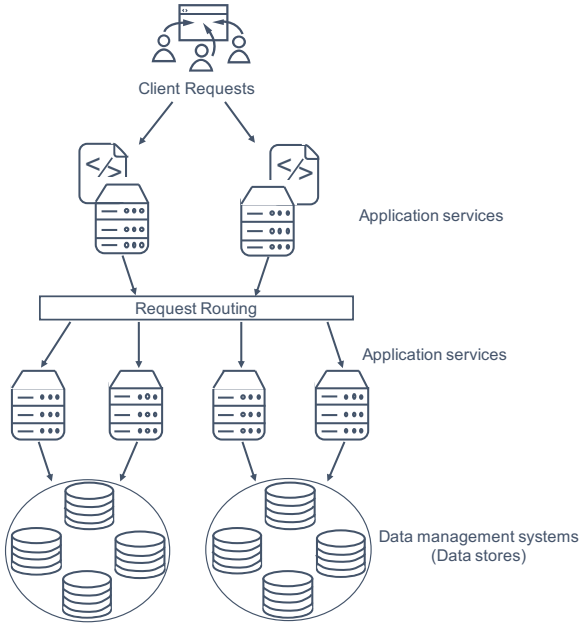


Figure 1.2: Abstract view of a typical modern scalable application comprising multiple micro-services. Stateful services generate their response by executing their processing logic on its state persisted on an external data store

and how to tune it for a particular application is a non-trivial undertaking even for experts.

To account for the need for scalable storage in terms of both capacity and performance, modern data storage systems are typically decentralized systems [5, 23, 32, 53, 70, 86, 132, 174]. Rather than relying on centralized storage arrays, distributed data stores consolidate large numbers of commodity servers into a single storage pool, providing large capacity and high performance at low cost over an unreliable and dynamically-changing (often cloud-based) infrastructure. Data is split into partitions and distributed across nodes allowing systems to scale in order to accommodate increases in client demands. Different forms of data replication [72] are typically used for reliability and high availability.

Scalable data stores are commonly used as a core component of the persistence tier of popular Internet-driven application services to maintain and manage their state while serving tens of millions users at peak times [86]. Figure 1.2 shows an abstract view of a typical modern scalable application designed by composition of multiple micro-services, as is often the case in enterprise software architecture today [109]. Stateful services typically

read and/or write state persisted in a scalable data store (bottom tier of Figure 1.2) as part of their processing logic before generating their response to client requests. While a scalable caching (middle) tier can reduce the amount of service requests faced by the scalable data store, writes must typically be handled by it in their entirety. As many applications have strict operational requirements in terms of performance, reliability and availability, state management systems often become a dominant factor affecting service quality [86].

The need for good performance from scalable data stores is often a business requirement. Large technology companies have highlighted the importance of systems performance in the success of any online venture [1, 6, 37, 41]. A data store needs to deliver its functionality with tight bounds in terms of I/O performance, usually described in terms of throughput (the number of requests served per second) and/or latency (time required to serve a request). Amazon reported 1% of sales loss for every 100ms of latency while Google experiences a 20% traffic drop for a 500 ms increased latency in search page generation [3]. Scalable data storage systems that can deal with huge data volumes in an efficient way, allowing applications to achieve their performance oriented goals and satisfy user requirements, are in need today.

While data store design for a specific target workload and environment/resources is an important goal by itself, the need to adapt to internal or external changes during the lifetime of a data store adds to the complexity of their design, as described next.

1.2 Adapting to internal or external changes

Scalable data store design can attempt to optimize for certain workload types by, for example, analyzing workload characteristics inferred from production traces [57]. As applications change over time however, workload characteristics may also shift leading to different transaction rates over time, access patterns, and so on. Data store designers today understand that they need to design for serving *dynamic workloads*, i.e. the workload characteristics such as data-access pattern and the load changes over time.

Data stores need to support on-line transactional processing (OLTP) applications on real-time data and on-line analytical processing (OLAP) tasks to support business intel-

ligence systems that mine large datasets for business insights. Traditionally application designers opt for data pipelines with separate state-management systems, optimised for different data processing tasks, leading to complex system designs and high operation costs. Modern data-processing systems handle *hybrid* workloads, combining real-time transactions and historical data analysis, known as hybrid transactional-analytical processing (HTAP) [92, 159, 169], as means to quickly transforming freshly obtained data into knowledge. Such systems require scalable data stores that are able to *adapt* to different workload characteristics. Change in the rate of client load is another dominant source of change in today's variable world (driven by sudden changes such as societal trends and news-driven flash crowds), that data stores should be designed to efficiently adapt to.

Adaptation is a requirement both for maintaining efficiency despite changes, but also to maintain performance guarantees expressed as *service-level objectives (SLOs)*, where applicable. A growing number of large-scale data stores used today (often as a service) by Internet enterprises are designed to offer performance guarantees expressed as SLOs. Guaranteeing a stable performance while the data store adapts to internal or external changes and transitions to new configurations is a challenging research area.

Adapting to unpredictable changes while hiding the intrinsic complexity of doing so from operators and end-users has long been the goal of a line of research aiming to achieve *autonomic systems* [124], with significant research activity starting in the early 2000s. Autonomic systems incorporate different degrees of self-management capabilities over distributed computing resources and commonly operate as monitor-analyze-plan-execute (or MAPE) control loops to support decision-making. Data stores are indeed long-lived systems facing significant challenges during their lifecycle. Just as any autonomic system, they should be able to sustain internal or external change; specifically, they should be able to adapt to changes transparently while maintaining efficiency or performance goals.

The need for adaptivity in data stores is not new. A form of adaptivity known as resource *elasticity*, namely the ability of a system to dynamically and in an online fashion adapt their service capacity to accommodate load fluctuations, has been a focus of research for some time [82, 88, 126]. A number of systems today offer elasticity features, meaning that they can dynamically adapt to load changes by increasing or decreasing their

resources during their lifecycle. Other systems employ decision-making techniques to decide on appropriate actions to take during initial deployment or under variations in workload and/or system configuration [122]. In this dissertation we build upon and extend previous work by studying challenges relating to multiple aspects of adaptation in the lifecycle of scalable data stores, not having been addressed so far. We additionally study the efficiency of existing adaptation mechanisms and propose improvements to them.

The challenges a data store (or any computer system for that matter) faces can be *internally* or *externally* induced. Examples of internal changes include periodic data reorganization, data checkpointing, or other data-management activities (such as garbage collections), known to be the source of performance variability, often resulting in violations of performance-oriented goals. An important example of external change is load fluctuations leading to the need for elasticity actions described earlier. External sources of performance overhead also include externally-triggered tasks such as data-backup activities or other external processing competing for resources with common-path processing in a data store. In addition, failures or workload changes may also be considered externally-induced changes. Ideally, the system should be able to automatically adapt to these changes in different ways: the use of *reconfiguration actions* to eliminate or hide their performance impact is one possible way that is explored in this dissertation.

Data stores often maintain and manage the state of stateful services (as depicted in Figure 1.2), cooperating with them within a multi-tier distributed system with well-defined boundaries, rather than a tightly integrated single software system. In such cases, different types of systems (the data store and the distributed middleware platform) are often *unaligned* in terms of their management policies, such as the allocation of resources and placement of tasks to those resources; better coordinating such system policies can improve cross-system communication and overall performance, for example by improving the locality of data access between processing tasks and data-store partitions. Often reconfiguration actions that are independently decided by one of the systems (such as elasticity actions) can be considered externally-induced changes that impact the degree of coordination previously achieved between the different systems. The need to better align (and maintain alignment over time) via cross-system coordination is another form of adapta-

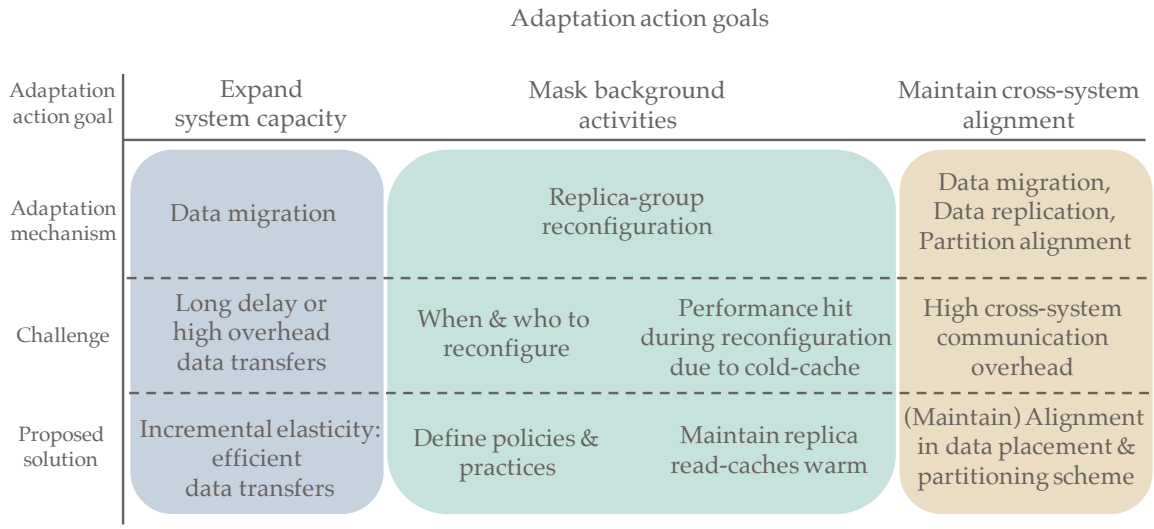


Figure 1.3: The goals and mechanisms of the adaptation actions used along with the challenges and the proposed solutions (contributions of this dissertation)

tion to external change that can lead to benefits in the overall system. Such adaptation actions should be performed in an efficient and low-cost way allowing for seamless transition to a new system configuration without any service disruption.

This dissertation focuses on research challenges that typically occur due to the aforementioned internal or external changes over the lifecycle of a distributed data store (a pictorial representation appears in Figure 1.3). The first adaptation action we focus on is the expansion of system capacity as workload demands increase or when the system tries to improve its performance by incorporating more resources. A second focus area is on adaptation actions aiming to mask the performance overheads of internal or external resource intensive background activities. We propose a reconfiguration action (replica-group reconfiguration) traditionally used for fault-tolerance in the context of hiding performance bottlenecks. We also identify performance inefficiencies on the mechanism itself and address them by proposing improvements over the mechanism. Finally, our third focus area is on continuous monitoring and improvement of data-access locality in distributed multi-tier middleware platforms that rely on distributed data stores to maintain and manage their state. We enhance data-locality in a cross-system fashion using existing reconfigu-

ration mechanisms such as data migrations, replication, and elasticity mechanisms and propose new methods for the alignment of the data partitions and the processing tasks.

Figure 1.3 summarizes the goals and mechanisms of the adaptation actions studied in this dissertation, along with the challenges and the proposed solutions, at the core of the contributions of this dissertation. Reducing performance variability and improving the overall system performance is a challenging task as the system transitions between configurations during its lifecycle. In the next section (§1.3) we provide a more detailed discussion on the challenges a data store faces in different phases of its lifecycle. In Section 1.4 we outline the contributions of this dissertation in addressing these challenges.

Thesis statement: This thesis experimentally proves that the aforementioned performance challenges during changes in distributed data stores can be addressed either via novel adaptation actions or improvements to existing ones.

1.3 Challenges and Assumptions

The major question we try to address in this dissertation is how a data store can improve its performance as it continuously adapts using certain adaptation actions to achieve the goals of its capacity expansion and/or to hide internal or external performance overheads during its lifecycle. However, we often face some major challenges. Adaptation actions come at a cost; systems have to pay additional overhead for their internal reconfiguration and data reorganization. We believe adaptation actions should be transparent to the users with minimal service disruption, ideally without any noticeable impact. Guaranteeing the continuous on-line operation of the system while it transitions to the new configuration with minimal performance impact is a major challenge exposing a key trade-off between the duration of the action and the performance impact. In general, the more aggressive an adaptation action, the deeper its performance impact while prolonged actions lead to delayed benefits of the system's adaptation actions. Ideally a system should be able to adapt to data growth or workload changes as quickly as possible to the dynamic service SLOs. Application designers often opt for ad-hoc solutions to mitigate these challenges; however, violating the application semantics [11, 40]. While our major focus is on the

system performance, we believe that any adaptation action should not compromise other aspects of the data store such as data availability, durability and consistency.

In this section we discuss the major challenges we face in this dissertation as well as our approach to address each one.

1.3.1 Expanding storage service capacity

Innovating applications often face the consequences of the rocket of popular success [91] while the emergence of Web 2.0, social networking, and the Internet of Things (IoT) increased the sources of new information resulted in a significant need for the storage of data. As modern data stores are distributed, consolidating large numbers of commodity servers into a single storage pool, they should be able to accommodate the strong data growth and allow new nodes to join (or leave) the system gracefully, with minimal performance and availability loss. Data-intensive applications therefore require data engines that are elastic, i.e., have the ability to expand or shrink the capacity on demand by provisioning the appropriate resources on an automatic manner to match, as closely as possible, the workload changes at any point in time [110, 111]. During peak times a store is required to be able to expand its capacity to match the applications' higher data access demand, often serving tens of millions of users [86, 91]. In addition to growing data capacity needs, applications demand high performance from the underlying data stores.

Elasticity in the context of data stores is not a new concept. The majority of large scale distributed data stores support elasticity actions, i.e. flexibility of the system to expand or shrink its capacity to match the demand. A premier example is DynamoDB [2], a proprietary scalable key-value store service offered by Amazon. While DynamoDB has been successful with applications that require fast and predictable performance, it will not scale automatically if the workload requirements change. Users should explicitly request more throughput. However, the effect of such a change request is not instant. Amazon states that the *"increases in throughput will typically take anywhere from a few minutes to a few hours"* [2]. A likely cause is that attempting to rapidly change a service-level objective may

[2]AWS DynamoDB Q&A documentation as of July 2018

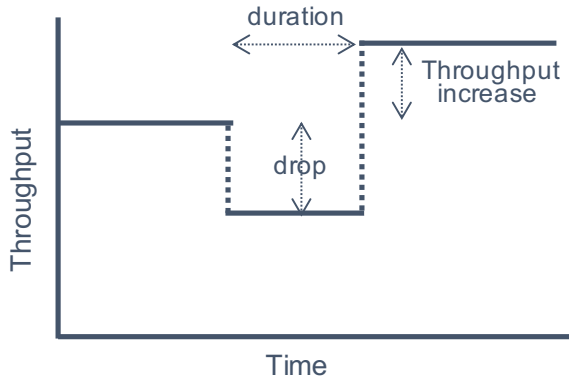


Figure 1.4: The performance characteristics ad-hoc data transfers during elasticity actions

pose an adverse impact to existing guarantees to applications

To better understand why such actions could impact the performance guarantees of a data store, we will describe the characteristics of a data store's expansion actions. The effect of elasticity actions is not instant as before the system being able to utilize the freshly joined nodes, large amounts of data have to be transferred to them. Data transfers over the network raise major performance challenges as they cause significant I/O activity. Figure 1.4 shows the performance characteristics of an elasticity action. Performing data transfers in an ad-hoc manner, results in an overall performance impact as all nodes that are engaged in data movements simultaneously reduce their processing capacity. In addition, new resources (joining nodes) cannot contribute their processing capacity until the time all data transfers are over while the many-to-all communication pattern in this phase is known, to under certain circumstances, be a cause of throughput collapse in large-scale data centers [157]. During elasticity actions there is a high performance-hit making the system unable to meet its performance objectives.

There is usually a trade-off between the duration of the adaptation actions and their performance impact during adaptation. Nevertheless, the more aggressive the adaptation action the deeper its impact on application performance. In this thesis we address the challenge of performing effective and low overhead data movements during an elasticity action while bringing new capacity to the system before the action is complete, thus, al-

lowing the system to achieve its performance goals. In Chapter 2 we propose *incremental elasticity*, a mechanism that orchestrates data transfers during elasticity actions. Incremental elasticity allows smoother transition to the new configuration as it ensures that fewer nodes are involved in network transfer at any point in time, reducing the overall performance drop during elasticity. In addition, as soon as a transfer is over, the associated data are becoming available for access on the new node, while a subsequent transfer of data takes place. We evaluate our mechanism under the widely used Cassandra key-value store, demonstrating that the system exhibits up to 2.6x fewer violations on its response time SLOs. The proposed mechanism is complementary to systems focusing on resource provisioning and systems capacity planning [182].

1.3.2 Hiding the overhead of internal or external background activities

During the steady state of a data store (when there is no need to expand the processing capacity or the performance goals of the store) the system performs internal data reorganization actions. Garbage collection actions such as LSM-tree compactions [22, 146], checkpoints and backup are necessary to guarantee a long term continuous and efficient operation of the system as well as data availability. These tasks are performed periodically and aim to minimize the space amplification and reduce the cost of read and write operations. Although data reorganization actions are performed as background activities allowing data availability, they result in significant I/O activity and high processing overhead which cause performance variation as the system serves client requests on the common path (right side of Figure 1.5). Thus the system may fail to guarantee specific service level performance objectives. Reducing the impact of such actions is challenging. Although there are approaches that try to model and estimate the cost of data reorganization activities [61] or even explore new mechanisms to reduce the impact [150, 163, 195], these tasks have still high overhead that impact the overall system performance. Ideally a system should combine the best of both worlds, the benefits of the data reorganization and the stable performance of a system that wouldn't need to perform these tasks.

As these actions cannot always be avoided or postponed for too long, the challenge

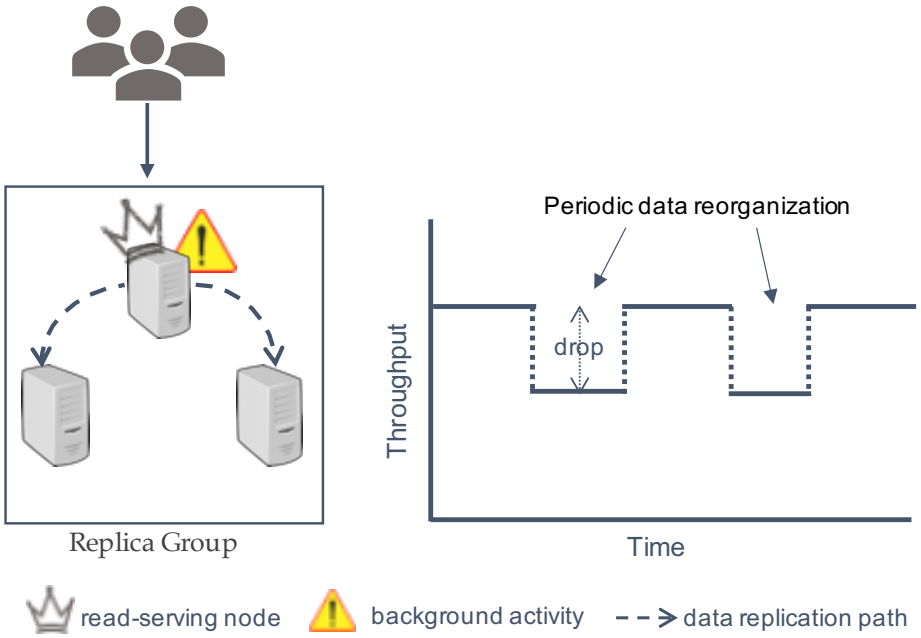


Figure 1.5: User requests served by a subset of nodes in replicated stores (e.g. stores following the Primary-Backup replication scheme, requests are served by the primary node) (left side). While being on the critical path the client-serving replicas typically take higher load than the other nodes. Any additional overhead on these nodes may critically affect performance of the replica group as a whole (right side of the figure)

we face is how to mask these overheads allowing the system to guarantee a stable performance while it performs data reorganization activity. To address this challenge, we follow a different approach.

Large scale distributed data stores use replication to improve data availability and durability. Dynamic reconfiguration of replica groups (i.e., the set of nodes that mirror the data) has received significant attention recently [62,137,147,168] as the importance of adjusting the number and type of nodes backing replica groups with minimal downtime has become a key system requirement of Internet applications. Clients usually direct their requests to a subset of nodes within the replica group [63,65,69,101,177]. However, being on the critical path of I/O activity, the client-serving replicas typically take higher load than the other nodes within the replica group. Any additional overhead on these nodes may critically affect performance of the replica group as a whole (Figure 1.5 left).

In this dissertation we show that lightweight replica group reconfiguration actions can be used as a means of adaptation actions to hide performance overheads allowing the system to exhibit a stable performance while it performs internal or external background activities. We redirect client requests away from background-activity busy nodes to hide the overheads imposed by the background tasks. In Chapter 3 we propose a lightweight adaptation action that allows replica-group members to dynamically change roles (i.e. primary replica, secondary replica) to hide the performance bottlenecks imposed by data store's internal and external background tasks; thus, eliminating performance variability. We describe the general design of such a system, evaluate multiple sources of overhead (LSM-tree [146] leveled compactions and data-backup tasks) over a wider range of workload mixes in a dedicated cluster, using an industrial-strength implementation based on MongoDB and an LSM-tree based storage manager, RocksDB [10].

1.3.3 Improving performance during replica-group reorganization

Data replication schemes are prevalent in data stores for high availability and reliability as data are replicated across a set of nodes, comprising a replica group. A range of existing replication techniques in distributed storage systems dictate how data are propagated to replicas [63, 65, 69, 101, 177]. However, there is a trend in practical systems to restrict reads to as few replicas as possible, to optimize for read-dominated workloads (Figure 1.6 left).

Reconfiguration actions may shift read-serving nodes within a replica group. A switch to the read-serving replicas may occur as a result of different types of reconfiguration operations. One such case traditionally occurs when a serving replica fails and a backup must take over (failover) or due to load balancing actions or workload migration where parts of an application are moved across the infrastructure. In a similar case, geo-replicated systems reconfigure the set of serving replicas (e.g. leader leader) while the client's location shift across time zones as they serve Internet users [53, 138, 166, 189]. Another reason for frequent (e.g. every 20 minutes) replica group reconfiguration include lightweight adaptation actions as a performance enhancement mechanism to mask performance bottlenecks on the serving nodes (Chapter 3).

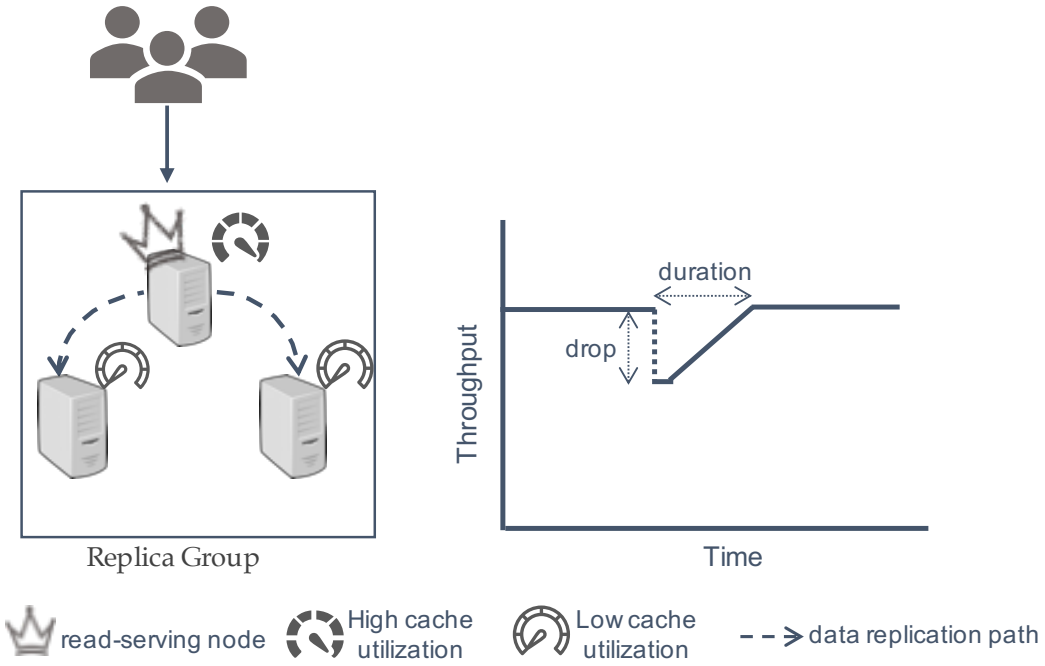


Figure 1.6: Right: Non-read-serving replicas do not fetch up-to-date cache contents in contrast to replicas serving reads Left: The cold-cache effect after a reconfiguration action that switches the read serving replicas. The new serving node requires time to recover to the pre-reconfiguration performance level due to cold-cache effect

Not surprisingly, data replication systems maintain in-memory data caches to improve performance, especially for read-dominated workloads. The efficiency of a read cache impacts the overall performance of data intensive systems. Data replication techniques focus on ensuring consistency of the persisted data set, while each node of the replica group also maintains a memory cache of that set to improve performance. Involving as few replicas as possible in read operations means that the caches of the remaining replicas receive delayed or no information about application reads, and thus do not fetch up-to-date content in memory, in contrast to replicas serving reads (Figure 1.6 left).

While replica group reconfiguration actions allow systems to recover after failures or aim to improve the overall performance, we observe a performance glitch during the reconfiguration action. A negative side effect at the time of switching the read-serving node, is a burst of cold-cache misses at the new read-serving replica. This may lead to latency

spikes and throughput drops that result in service-level objective (SLO) violations for significant amounts of time (several minutes) (Figure 1.6 right).

In this dissertation we propose a mechanism that allows replica nodes to get prepared for the forthcoming replica-group reconfiguration. Our study in Chapter 4 shows that the system exhibits up to 70% hit after a reconfiguration due to cold cache misses, taking almost 18 minutes in some cases to fully restore to the pre-reconfiguration level of performance. To address this issue we propose a new mechanism that allows the system to maintain up-to-date read caches across replicas without affecting the data consistency and availability properties of the system. Our method eliminates the cold-cache effect observed after a reconfiguration on the new node that serves reads, which impacts the overall system performance observed by the clients. We describe a lightweight, best-effort synchronization method that tries to minimize the overall cache divergence between the primary and replica nodes allowing for a seamless transition to the new configuration. The system exhibits a seamless transition to the new configuration avoiding downtime and an underperforming period.

1.3.4 Aligning data store partitions with distributed middleware tasks

In multitier architectures distributed middleware systems rely on data stores to maintain and manage their state. As these systems usually have strict performance requirements, the underlying data stores need to deliver its functionality with even tighter bounds. Tuning both systems is a challenging task. Determining how to effectively combine and coordinate the two types of systems remains a challenging research problem, as has also been pointed elsewhere [181]. Typically, the data store and the middleware –as two different types of systems, have a different lifecycle. Distributed systems are deployed over a set of nodes. The lack of coordination across systems results to unnecessary network cross-talk communication and misalignment of adaptation policies (e.g. elasticity). Even when different systems are deployed on the same set of nodes (e.g. the data store is colocated with processing tasks), this does not guarantee an efficient systems alignment.

In this dissertation we use stream-processing systems (SPSs) as a distributed middle-

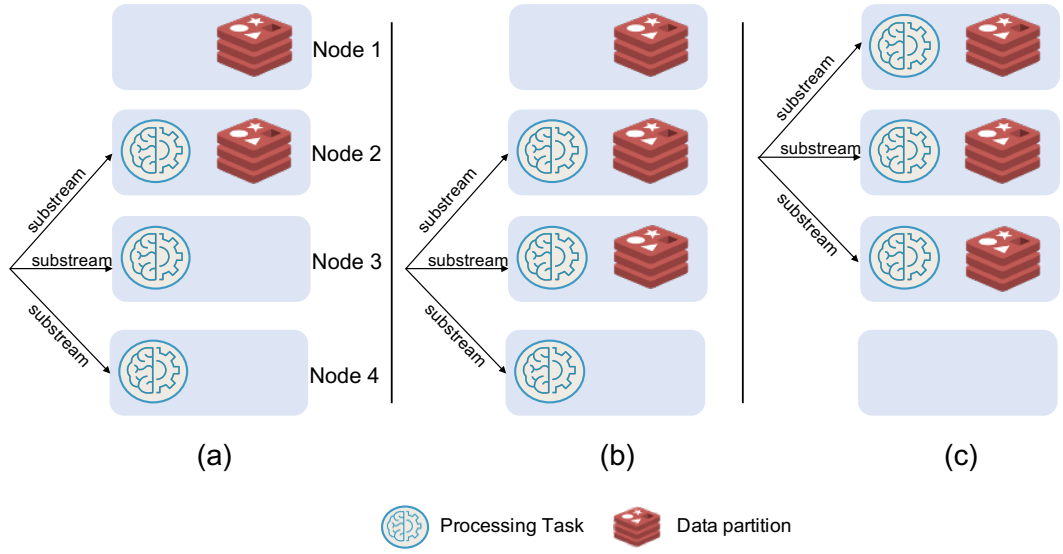


Figure 1.7: Amoeba alignment plan example: (a) Initial deployment (3 parallel processing task instances access 2 data partitions), (b) creation of new data shard, (c) task migration and adaptation of the partition-scheme

ware platform use case. Scalable stream-processing systems and analytics applications provide valuable insights by processing trillions of events per day [21]. Persistence of the applications external state, such as data produced by streaming jobs or by other applications or systems or ground-truth table data that can be accessed during the execution of analytics applications, is an important concern. Typically a data store's lifecycle is disconnected from that of the analytics applications. The cross-layer communication (between the application and the data store (Figure 1.2) highly affects the overall performance. The lack of coordination between systems and misalignment of adaptation policies (e.g. elasticity actions) misses opportunities for better cross-layers communication allowing for an overall improved performance.

We build upon the idea that the data access path can be improved via co-location of the processing tasks and data partitions. In Chapter 5 we describe the design and implementation of Amoeba, a system that coordinates the alignment between stream processing task partitions and data store partitions. Each system may apply a different initial deployment topology and parallelism (1.7(a)). Amoeba generates alignment plans that can the

align that parallelism of systems (Figure 1.7(b)) and/or migrate data partitions or processing tasks across nodes (Figure 1.7(c)) to improve data locality. However, co-locating both systems in the same set of nodes does not always lead to efficient cross-system communication. Amoeba can also adapt the systems partitioning scheme to exploit the co-location between processing operators and data partitions We also demonstrate the decision process during the alignment plan generation. We experimentally evaluate Amoeba on an advanced, complex stream processing benchmark application, Linear Road [52, 180], on large-scale deployments of up to 64 nodes on Amazon EC2 platform [3] showing 2.6x performance improvement when Amoeba aligns the data store with the SPS tasks.

1.4 Contributions

The goal of this dissertation is to propose novel adaptation mechanisms and improvements on existing ones aiming to improve the overall performance while the data store adapts to internal or external challenges. More specifically the contributions of this dissertations are:

1. We study the performance impact of data transfers over the network during the elasticity actions when the store expands its capacity. We propose a new mechanism that schedules data transfers in a fine-grain manner reducing the performance overhead of data transfers while progressively increases the processing capacity of the system in an on-line incremental fashion. The proposed method shows early benefits of data transfers during the elasticity action as it incorporates new resources and makes data sub-sections available prior to completing the full data transfers. In our evaluation we use the widely used Cassandra key-value store [132] and demonstrate 2.6x fewer violations on its response time SLOs during the elasticity actions.

2. We propose replica-group reconfiguration as a way to mask performance bottlenecks in replicated data stores following the primary-backup replication scheme. We investigate the benefits of changing replica-group leadership prior to resource-intensive

[3] https://aws.amazon.com/ec2

background tasks (e.g. LSM-tree compactions [146] or data backups on the primary node). We apply our mechanism in a state-of-the-art store, MongoDB [23] using RocksDB [10] as its internal storage engine, demonstrating that our adaptation mechanism is lightweight and can effectively hide performance bottlenecks of internal or external background activities allowing the system to guarantee a stable performance.

3. We study the impact of reconfiguration actions within the replica groups. Starting from the observation that under certain circumstances practical systems restrict reads to a few replicas (typically one) to optimize for read-dominated workloads, we observe a performance glitch during the reconfiguration as secondary replicas cannot fetch up-to-date contents into their own cache. We address this issue by proposing a new mechanism to maintain up-to-date read caches across replicas without affecting the data consistency and availability by disseminating read-hints within replica-group. Our proposed solution is a lightweight, best-effort synchronization method that tries to minimize the overall cache divergence between members of the same replica group. Thus, the system is able to seamlessly transition to the new configuration without the performance gap during the replica group re-organization. This is especially important under the read-intensive workloads that are common today.

4. We investigate the benefits of automatically aligning data stores with distributed middleware systems that rely on data stores to maintain their state. We describe the design and implementation of Amoeba, a system that continuously strives to discover alignment opportunities across systems and improve data locality. Our system achieves this either via appropriate (informed) initial placement of data partitions and alignment of data partitioning schemes across different systems with the ensuing data movement where appropriate while it continuously coordinates adaptation actions across systems. We experimentally evaluate Amoeba on an advanced, complex streaming benchmark application, Linear Road [52, 180], on large-scale deployments of up to 64 AWS nodes showing 2.6x performance improvement when Amoeba aligns the data store with the SPS tasks. Our novel solution addresses the

research problem of automatically streamlining and integrating state management capabilities across stream and storage technologies, a complex undertaking that today requires deep human expertise, making it a hard and failure-prone undertaking.

Figure 1.3 shows the goals and the mechanisms of the adaptation actions we study in this dissertation along with the challenges and the proposed solutions (contributions of this dissertation) The dissertation has different types of contributions that can be grouped based on common characteristics as follows:

- **Improvements to existing adaptation mechanisms**. More specifically, this thesis contributes to more efficient elasticity actions and replica-group reconfiguration actions (contribution 1 and 3)
- **Studying novel adaptation mechanisms**. We propose replica-group reconfiguration as a performance enhancement adaptation action (contribution 2). We also propose data partitions and processing tasks alignment through the adaptation of the partitioning scheme (contribution 4).
- **Using of known adaptation mechanisms** (migration, replication, elasticity) **to improve efficiency** in multi-tier (distributed middleware - data stores) systems (contribution 4)

The space of large scale distributed storage systems is broad. The design, implementation and evaluation of our proposed mechanisms that address the challenges largely fall in the category of NoSQL key-value stores. Our implementation and evaluation effort in this thesis is based on and in certain cases extends several state of the art, widely used systems in this area: Cassandra [132], MongoDB [23], RocksDB [10], Redis [32]. However, the contributions of this dissertations have a broader applicability in the area of distributed storage system that share the same design principles, namely replication (i.e. multiple copies of data stored on different locations to improve the overall data availability and durability), sharding (i.e. split data in multiple partitions that are held on separate store instances to spread load) and elasticity (i.e. the system is able to expand or shrink its capacity on demand). To the best of our knowledge, the majority of modern data storage

systems follow these design principles and thus the work of this dissertation can be more broadly applicable.

1.5 Related work on autonomous storage systems

The concept of storage systems that can adapt to application requirements is not new. In the last five decades, storage systems have evolved significantly in this direction. One way for storage systems to enable adaptation is by exposing configuration knobs that allow users to access the underlying (re)configuration mechanisms and tuning their behaviour according to performance objectives. However, optimizing data stores to meet the applications needs is a tedious and complex task even for experienced system administrators.

The idea of automatically tuning storage systems through optimal configuration settings allowing them to adapt to workload characteristics, available resources and other challenges they face, was initially explored in the 1970s. In the late 1970s, the idea of *self-adaptive* systems was proposed [106]. They were meant to be used as recommendation tools aiming to assist the database physical design (such as the selection of indices to match projected access requirements) [107, 108, 136, 194]. Later, during the late 1990s and early 2000s, *self-tuning* systems [73, 188] supported additional recommendations for multiple data store configuration knobs (e.g. configure the database memory heaps and cumulative database memory allocation) [74, 85, 172, 173, 185].

Early systems were created as standalone and external to the data store systems, used as recommendation tools to assist system administrators in the tuning process. As the number of the data store configuration knobs increased significantly, some of the configuration tools included the support of automatic knob configuration through the data stores APIs. The reconfiguration settings are based on ”best practices” and heuristic algorithms evaluating hard-coded rules in order to discover the optimal configuration settings [108, 179, 185]. Heuristic-based approaches rely on assumptions regarding the workload and the environment of the storage system that may not always accurately reflect the real-world. More recent knob configuration tools are built using machine learning techniques [186]. These techniques are based on model training using runtime collected data.

In addition, this approach allows the system to leverage past experience and reuse training data gathered from previous sessions [186].

In this approach the reconfiguration policy is external to the data store, defined in the external recommendation systems. Users or external systems treat the data store as black box leveraging the reconfiguration mechanism through the exposed knobs and APIs to apply the reconfiguration policy. The data store is not aware of the reconfiguration actions plan and the targets of the policy or the applications goals.

Another approach is to define the reconfiguration policy and the goals of the systems *within* the data store [183,184]. The data store does not necessarily expose reconfiguration knobs for the underlying reconfiguration mechanisms that are integrated into its core. The reconfiguration actions do not require any human intervention or external systems that monitor the data store and discover tuning opportunities through reconfiguration actions. The data store continuously monitors its environment and workload characteristics and decides to reconfigure itself to adapt to its internal or external challenges [156]. Recent work in this direction focuses on the adaptation of the internal data representation [48,55] and tune its internal data structures [83,84].

This thesis contributes in a number of ways towards more autonomic data stores applying reconfiguration actions to improve overall system performance, leveraging existing reconfiguration mechanisms as well as proposing new ones. The reconfiguration policy is integrated into the data store whenever possible, allowing it to reconfigure itself as it faces internal or external challenges without any external input. However, in cases where the data store has to coordinate its reconfiguration actions with the (re)configuration actions of other systems –such as in multi-tier architectures where the cross-system alignment is necessary to improve the overall performance– external reconfiguration systems are necessary to orchestrate the reconfiguration actions of the data store.

Overall, in this dissertation we contribute towards the vision of autonomous data stores incorporating different reconfiguration policies and mechanisms that allow efficient adaptation actions addressing internal or external changes throughout their lifecycle.

1.6 Outline of Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2 we study the performance impact of data movements during elasticity actions. We identify that performing data migrations in an ad-hoc manner overloads all involved nodes leading to overall system performance degradation. To address this problem we propose a new mechanism, incremental elasticity, that orchestrates data movements to reduce the performance impact. The fine-grain data migrations allow the associated data become available for access on the new node while the whole elasticity action is still in progress.

In Chapter 3 we study the impact of data stores' internal or external background activities competing for resources. We discover that these tasks are the source of performance variability. To overcome these challenges we propose a lightweight adaptation mechanism that is based on replica-group reconfiguration by redirecting the client requests away from nodes that perform background activities.

In Chapter 4 we study the performance impact during replica-group reconfiguration actions. We discover that the system performance suffers from high cache-miss rate on some nodes right after the reconfiguration as practical systems restrict read requests only to a subset of nodes. As a result, non-serving replicas cannot keep their in-memory cache contents updated; thus, during a reconfiguration or a fail-over action, the system suffers from a high read-performance impact for a significant amount of time due to cold-cache misses. In the chapter we describe our approach to address this problem. We propose a mechanism to maintain up-to-date read caches across replicas by disseminating read hints to the non-serving replicas to keep their caches warm; thus the system is able to seamlessly achieve the same performance level even in the face of a replica group reorganization.

In Chapter 5 we study the benefits of automatically aligning data stores with distributed stream processing systems that rely on data stores to maintain their state. We propose Amoeba, a system that dynamically adapts data-partitioning schemes and/or task or data placement across systems to eliminate unnecessary network communication across nodes. The system incorporates adaptation actions such as data migrations, elasticity actions

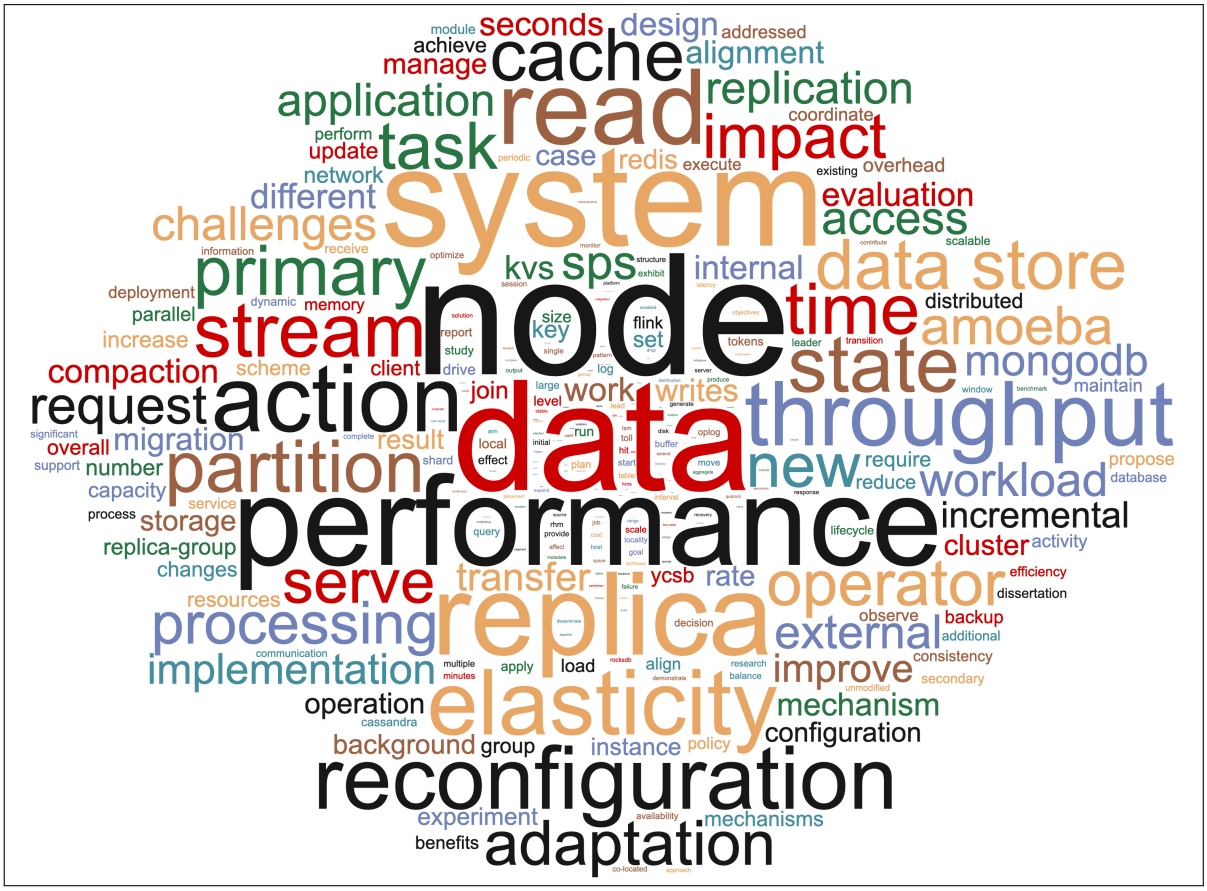


Figure 1.8: Word cloud based on the text of this dissertation

and partitioning scheme alignment across systems to improve the overall systems performance.

Finally in Chapter 7 we summarize and conclude and in Chapter 8 we point directions for future research

Figure 1.8 presents a *word cloud* a visual representation of the text of this dissertation (*word cloud*). Words that appear bigger are more often mentioned in the text.

Chapter 2

Incremental elasticity

The emergence of Web 2.0, social networking, and the Internet of Things increased the sources of new information and resulted in a significant need for the storage of data. Although part of the data may be stored temporarily for subsequent analysis, storage space and throughput requirements rise dramatically during time periods where large amounts of new data is created. Data-intensive applications therefore require data engines that are *elastic* (can adapt resources to requirements dynamically), accommodating strong data growth and allowing new nodes to join (or leave) the system gracefully, with minimal performance and availability loss.

In this chapter we study elasticity mechanisms in distributed data stores and the performance impact while the systems transition to the new configuration. A number of large-scale data stores in the marketplace today used (often as a service) by Internet enterprises are designed to offer performance guarantees expressed as *service-level objectives*. A premier example is DynamoDB [2], a proprietary scalable key-value store service offered by Amazon. While DynamoDB has been successful with applications that require fast and predictable performance, it will not scale automatically if the workload requirements change. Users should explicitly request more throughput. However the effect of such a change request is not instant. Amazon states that the "*increases in throughput will typically take anywhere from a few minutes to a few hours*" [1]. A likely cause is that

[1] AWS DynamoDB Q&A documentation as of July 2018

attempting to rapidly change a service-level objective may pose an adverse impact to existing guarantees to applications. This means that throughput spikes cannot be rapidly remedied. As a result, applications using DynamoDB may experience periods where a portion of their requests will be dropped or blocked until the system is reconfigured. This leads application developers to opt for ad-hoc solutions [11,40] which may violate application semantics (e.g. violate durability semantics when using intermediate buffers/queues in order to handle request throttling by DynamoDB).

There is usually a trade-off between the duration of the elasticity actions and their performance impact during adaptation. Ideally, the system should be able to adapt to data growth or workload changes as quickly as possible to satisfy application requirements. Nevertheless, the more aggressive the adaptation action the deeper its impact on application performance. In this chapter we focus on the performance impact of elasticity actions in NoSQL data stores. In particular we target data stores that horizontally partition data (referring to data partitions as *shards*) and spread them as far as possible on the available nodes for load balancing.

Next we will give an overview of elasticity actions in modern distributed NoSQL stores. Several popular data stores [5, 86, 132, 175] require a new (joining) node to take responsibility for sections of data spread across the existing nodes to achieve a more efficient load balancing, receiving the corresponding data via network transfers. One common option to carry out these network transfers [5, 132, 175] is to perform them in parallel towards the new node, raising a number of challenges: First, a large number of nodes are simultaneously reducing their processing capacity while engaged in data transfer, resulting in an overall performance impact. Second, the new (joining) node is solely engaged in data transfer (at the speed of its network link) and cannot contribute processing capacity until the time all data transfers are over. Third, with all data transfers completing at about the same time, associated activities such as data compactions [146] are likely to also overlap, resulting in significant I/O activity at the new node during the early stages of its normal operation [8]. Fourth, the many-to-all communication pattern in this phase is known to under certain circumstances be a cause of throughput collapse in large-scale data centers [157]. In this chapter we describe *incremental elasticity*, our proposed mechanism

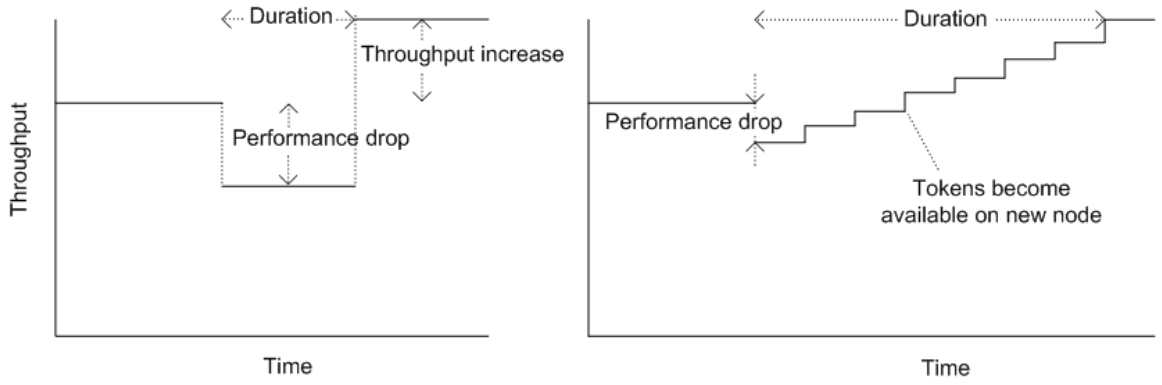


Figure 2.1: Performance characteristics of parallel network transfers (left) vs. incremental elasticity (right)

that addresses all four challenges.

Incremental elasticity replaces parallel network transfers with a sequential communication schedule where senders take turns sending data to the new node. As soon as a transfer is over, the associated data are becoming available for access on the new node, while a subsequent transfer of data takes place. The expected performance benefits of incremental elasticity (visualized in Figure 2.1) are summarized as follows: Fewer nodes are involved in network transfer at any time, reducing the overall performance drop during elasticity. With data becoming available on the new node as soon as data transfers complete, processing capacity increases in a step-wise incremental fashion, providing early benefits of the newly available capacity while the elasticity action is still on. With only a single sending server active at a time, the delay due to its higher load may be masked by other replicas of the data it owns.

In this chapter, we demonstrate the benefits of incremental elasticity using Cassandra [132], a popular open-source key-value store inspired by Dynamo [86]. Our contributions are:

- A new mechanism for gradually increasing the processing capacity of scalable data stores called incremental elasticity

- An implementation of incremental elasticity in the context of the Cassandra column-

oriented key-value store

- An experimental evaluation of the benefits of incremental elasticity vs. simultaneous parallel network transfers under Yahoo! Cloud Serving Benchmark (YCSB) workloads

The remainder of this chapter is structured as follows: Section 2.1 provides background on elasticity mechanisms in data stores, including specifics on Cassandra. Section 2.2 describes our design and implementation of incremental elasticity. Section 2.3 describes the performance results and comparison of incremental elasticity vs. parallel transfers in Cassandra under different YCSB workloads. Section 2.4 discusses related work and finally we summarize our conclusions in Section 3.4.

2.1 Elasticity mechanisms in data stores

Elasticity is the ability of a system to dynamically (in an online fashion) adjust its processing capacity by adding or removing resources to meet workload changes. Changes to system capacity while a workload runs typically impact its performance. Ideally, data store elasticity should result in an adjustment of processing capacity proportional to the resources being added or removed, completes at the shortest possible time, and impacts running workloads as little as possible. To achieve a proportional increase in overall performance, the data migrated should be chosen appropriately so as the elasticity action results in a well-balanced system. Elasticity may either expand capacity in a *horizontal* manner, adding or removing nodes from a cluster, or in a *vertical* manner increasing or decreasing the amount of resources (CPU, memory, I/O) of existing nodes of a cluster. In this thesis we focus on the most common form of elasticity actions, the horizontal elasticity, in data stores whose nodes privately own and manage their storage resources (local or network disks or SSDs). Thus, a new (joining) server must receive the data it manages to its own storage upon joining the cluster, via network transfers from other data-store nodes. In such a model, elasticity actions apply to both compute and storage resources.

Distributed data stores split data into partitions, also termed *shards*, and assign them

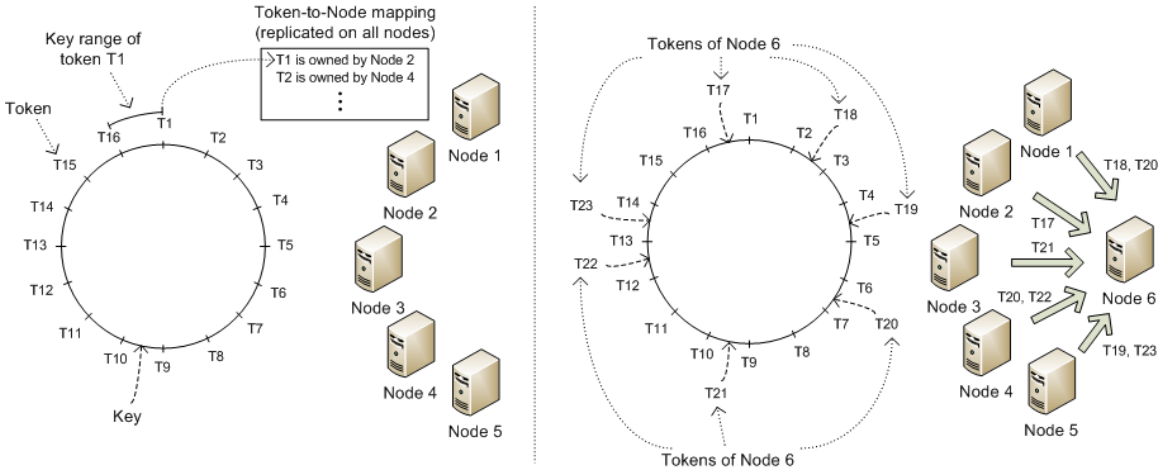


Figure 2.2: Creating shards through tokens (left); cluster expansion via parallel network transfers (right)

across system nodes. There are several different methods for data partitioning. Several popular data stores [5,86,132,175] achieve data partitioning using consistent hashing [119] for mapping keys to nodes. The specific variation of consistent hashing used in this work (implemented by the Cassandra [132] column-oriented data store) is to associate each physical node with a number of non-contiguous key ranges, *tokens*, hashed to a ring (Figure 2.2 left). Each token identifies a *key range*, starting from the previous token (in counterclockwise order) up to it. Tokens are also referred to as "virtual" nodes or *vnodes*. Keys are mapped first to tokens and then to nodes. This allows a finer granularity in data distribution among servers [18]. Cluster expansion via the addition of a new node introduces a number of new tokens to the ring (e.g. tokens of Node 6 in Figure 2.2 right). The new node will take responsibility for key ranges identified by its tokens, and will receive the corresponding data via network transfers from previous data owners (Figure 2.2 right).

To durably store data, Cassandra implements a version of Log-Structured Merge (LSM) trees [146]. Cassandra nodes apply each incoming update to a write-ahead log for durability and then to an ordered memory buffer (the memtable), which is periodically flushed to an indexed ordered file (the SSTable). A number of SSTables may be consulted to retrieve a requested key/value. Periodic compactions (merge sort runs) of SSTables aim to maintain

a small number of large SSTables, improving read performance.

Elasticity via parallel data transfers. The Cassandra column-oriented data store performs elasticity actions by transferring data to a joining node via parallel network transfers. The associated configuration (state) changes are communicated via a gossip protocol [132]. At startup, a new (joining) node uses the gossip protocol to announce itself to the cluster and to receive information about the node topology as well as the tokens that each existing node owns in the cluster. The startup phase of a joining node is also referred to as *bootstraping*. An important task during this phase is to select the tokens for which the new node will take responsibility, and to initiate the data transfer corresponding to these tokens from existing cluster nodes. Fine granularity in data distribution ensures that a new node will assume responsibility for a balanced share of data from other nodes in the cluster [18]. All data transfers are performed in parallel.

Cassandra uses a *streaming* protocol to handle the data exchange among nodes in the cluster. Bootstrap of a joining node involves several stream sessions, each involving the joining node (*receiver*) and one of the existing nodes in the cluster (*follower*), transferring data between each pair of nodes in four steps: (1) The receiver initiates a stream session with a follower; (2) After the initial handshake, the receiver sends a list of tokens (key ranges) it needs from the follower. Upon receiving the preparation message, the follower records which sections of the data it owns (the SSTable files) it has to send according to the requested tokens and enters the streaming phase; (3) During the streaming phase the follower sends the data to the new node; (4) Finally after all the data transfer tasks complete, the nodes agree to close the session.

During bootstrap, the joining node does not serve client requests. To maintain availability during elasticity actions, Cassandra does not suspend updates on tokens (key ranges) involved in streaming and about to change ownership. To ensure that the joining node will receive such updates, Cassandra operates as follows: Prior to initiating streaming, the bootstrapping node announces the tokens that it will be responsible for. Existing nodes in the cluster mark these tokens as *pending* and forward a copy of any write request on the pending tokens to the joining node (in the following we refer to these writes as *shadow*

writes). In this way, when the new node finally joins the cluster, it will already have received any updates accepted by the system after streaming started, avoiding consistency gaps that may lead to repair [31] operations in the future. When all streaming sessions complete and all data are available to the new node, it updates its status from *joining* to *normal* state and announces (through gossip) its availability to serve client requests. Note that token transfers typically materialize as many small SSTables in the new node, triggering several data management activities such as data compactions to consolidate them.

The aggregate ability of all sending nodes to transfer data may exceed the available network throughput at the receiving side. TCP will thus appropriately throttle each sender. Additionally, Cassandra offers a configuration option (*stream_throughput_outbound_megabits_per_sec*) to limit each sender's streaming throughput. This addresses an empirical observation (e.g. [130]) that high streaming throughput often leads to significant performance variability. Throttling sources reduces the impact of streaming on performance, however it would also increase the duration of elasticity without any incremental benefit in the meantime, as all tokens are made available only at the end of the streaming sessions.

2.2 Design and implementation

Design. At its core, incremental elasticity orchestrates data transfers in order to bring new capacity earlier into the system and reduce the overhead of ad-hoc data movements. It involves a communication mechanism that transfers data to a joining node via a sequential schedule where senders take turns sending data to the new node. As soon as a transfer is over, the associated data are becoming available for access on the new node, while a subsequent transfer of data is in progress. To enable incremental elasticity, senders should either cooperatively decide the schedule of data transfers or assign the task of coordinating those transfers to the joining node, aiming for sequential transfers between a pair of nodes (existing node, new node), allowing only one active such session at a time. Generally speaking, we can schedule transfers between arbitrarily-sized subgroups of nodes and the joining node but we consider only subgroups of size one in this work.

Assuming the simpler solution that the joining node is responsible for coordinating

data transfers in a pair-wise manner, the choice of which node to stream from can be simple, such as using a round-robin policy or random. Other policies however, such as starting from nodes that are better prepared for the transfer, are possible. The data transfer protocol used should support pair-wise transfers and orchestration by the new (receiving) node, as can (with appropriate modifications) the streaming protocol described in Section 2.1.

At startup the new (joining) node should build a current view of the system by contacting its membership service and receiving information about the node topology and data distribution in the cluster (Figure 2.2 left). At this point it must be decided which parts of the data the joining node will eventually serve. A key design point for incremental elasticity is to ensure announcements of data ownership (e.g., that a certain region of data is now served by the new node) are now performed incrementally and in a piecemeal fashion. This affects both read and write operations: As soon as a section of data is now owned by the joining node, reads are now directed to it, and writes need no longer be performed twice (cf. *shadow writes* described in Section 2.1).

Under parallel data transfers, a joining node is fully dedicated to processing stream sessions as fast as its CPU resources are able to cope with network traffic (ideally at link speed). As the aggregate network bandwidth of all senders may far exceed the network bandwidth of the receiver, senders are expected to be throttled by TCP, if not by data-store-level streaming limits. Replacing parallel stream sessions by sequential ones should not necessarily increase the overall duration of the elasticity action, if data transfer still happens at the speed of the receiver's network link. In incremental elasticity, the same streaming throughput could be achievable even with a single sender, if the sender is able to send as fast as its network link allows it. While we initially aimed to achieve the same duration of elasticity actions between parallel and incremental elasticity, in our implementation we found that senders cannot achieve full network speed, even with the addition of multithreading techniques. However, reducing the speed at which the joining node processes incoming traffic means that the node has more CPU resources to devote to processing requests after the first batch of tokens become available. Any delay in completing the elasticity action is thus compensated by additional processing capacity available during the

action. Even if we are able to (through a more aggressive implementation) achieve full link speed at the receiver, acquiring additional processing capacity there can be achieved by dynamically changing the allocation of memory and virtual CPU resources at runtime through most hypervisors, for the duration of elasticity.

We consider the above design principles as straightforward and believe that they can easily be supported by many distributed data stores. In what follows we describe our implementation in the Cassandra column-oriented data store.

Implementation. Our implementation extends Apache Cassandra version 3.7, especially its gossip and streaming components. Each Cassandra node has a *StreamManager* module responsible for all streaming operations. When a new node bootstraps it creates a *StreamPlan* object to associate token requests with the existing nodes. Internally it builds *StreamSession*s to handle network communication between nodes. The *StreamPlan* is associated with a *StreamCoordinator* component that manages the *StreamSession*s. *StreamCoordinator* initiates the stream sessions sequentially ensuring that only one is active at a time. As soon as a stream session is over, it selects the next session to start from the inactive stream sessions queue.

The state of a node indicates its status in relation to the cluster. At startup when the joining node announces itself to the cluster, it enters the *joining* state. In this state, it does not receive any client requests (either directly from clients or re-directed from other nodes). State transitions are announced using Cassandra's gossip protocol. With incremental streaming we expect that the aggregate processing capacity of the cluster should increase each time a batch of tokens are made available to the new node. In our implementation we introduce a new state, the *partial join* state, for the joining node. As soon as the first streaming session is over, the bootstrapping node enters in state *partial_join* in which it remains until the end of the streaming process. This state indicates that the node has available data and is able to serve client requests on a subset of tokens but is not yet fully integrated to the cluster. To support the *partial_join* state, we introduced a new type of gossip message sent by the new node when a stream session completes, announcing the state and informing the cluster of the tokens the new node is responsible for. Nodes

receiving this message update their tokens-to-node mapping, and start to appropriately redirect client requests to the new node.

Modifying the scheduling of token transfers means that we should modify the scheduling of *shadow writes* as well. In incremental elasticity, shadow writes only concern tokens that are being streamed at a specific time period. In standard Cassandra, nodes mark as pending those tokens that the joining node advertises as tokens it will eventually be responsible for. In our implementation, only tokens currently being transferred (streamed) are shadow-written until the end of the current streaming session. Tokens that correspond to future streaming sessions are fully served by their current owning nodes and there is no need to be shadow written. This is possible since the new node announces the pending tokens during the preparation step of each stream session and tokens become available when the session is complete. Overall this reduces the load that shadow writes place on the joining node during streaming. As soon as all data transfers are complete, the new node updates its state as normal and announces it to the cluster.

2.3 Evaluation

2.3.1 Experimental testbed and methodology

Our experimental testbed is a cluster of 9 servers, each equipped with a dual-core AMD Opteron 275 processor at 2.2GHz with 12GB of main memory. All servers run Ubuntu 14.04 64-bit with a 3.14.1 Linux kernel and are interconnected via a 1Gb/s Ethernet switch. Servers store data on a dedicated 300GB 15,500 RPM SAS drive that delivers 120MB/s in sequential reads and 250 IOPS in random reads with a 4K block size. Hard drives are formatted with the ext4 file system.

We use Cassandra version 3.7 with the OpenJDK 1.8.0-91 Java runtime environment. Our evaluation workload is the Yahoo Cloud Serving Benchmark (YCSB) [81] version 0.11 executing on a dedicated server. The benchmark is configured to produce two different mixes of reads vs. updates/writes: 95%-5% (Workload B) and 50%-50% (Workload A) respectively, with requested keys selected randomly with the Zipf distribution. The YCSB

evaluation dataset consists of 80 million unique records, replicated 3 times resulting to 240 GB of data (34GB per node) when loaded. Our initial Cassandra cluster consists of 7 servers. During elasticity actions an 8^{th} server joins the cluster.

We experiment with Cassandra data-consistency levels QUORUM and ALL. QUORUM requires responses from a majority (2 out of 3 in this case) of replicas to complete a read or write operation, whereas ALL involves all replicas. In QUORUM reads, 2 of the 3 replicas return only a checksum of the data, a Cassandra optimization to reduce network traffic [17].

The number of YCSB client threads (number of parallel connections between database client and servers) is set to 30 and 25 for the QUORUM and ALL consistency levels respectively, empirically determined to stress the cluster while keeping average response time under 50ms (considered a reasonable threshold). We used the default settings for node caches, namely key cache enabled and row cache disabled (however nodes benefit from caching at the OS buffer cache). Unless stated otherwise, we do not limit each node's streaming throughput (stream_throughput_outbound_megabits_per_sec set to 0). For repeatability, we modified the default token partitioner to ensure that each Cassandra node will be responsible for serving the same key ranges across experiments. The dataset is loaded fresh onto Cassandra nodes before each experiment.

Impact of compactions. To isolate the performance impact of SSTable compactions on the joining node from the impact of the streaming process itself we chose to disable compaction activities on the joining node (node 8) during its bootstrapping process. This choice was based on the observation that high compaction activity is significantly penalizing the joining node right after the elasticity action under parallel streaming. It also reflects standard practice in field use of Cassandra [8]. We observed that under incremental streaming, the impact of compactions is spread over time, however for fairness we used the same delayed compaction policy in that case as well. Other solutions to reducing the impact of compactions on the joining node are to limit compaction throughput and the number of active compaction tasks at any time, however an in-depth investigation of this aspect is beyond the scope of this work.

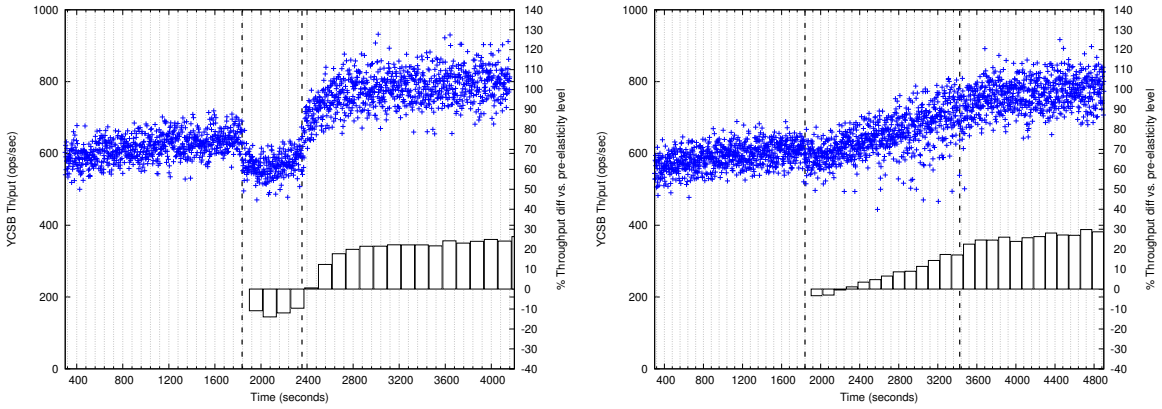
2.3.2 Analysis of experimental results

Figure 2.3 depicts YCSB throughput (YCSB ops/sec) with the workload mix for 95% reads 5% writes and QUORUM consistency. The elasticity action is triggered 30 minutes into the experiment when the system is deemed to have reached a steady state. The duration of streaming activities is highlighted by dashed vertical lines. In the inset we depict a bar chart indicating the relative performance change during the elasticity action vs. pre-elasticity performance. The latter is the average throughput during the most recent 5-minute time window before the elasticity action starts. Each histogram bar corresponds to an average over a 2-minute interval window.

Figure 2.3a (parallel streaming) exhibits a clear performance hit of about -14% during data streaming, followed by a performance increase (vs. pre-elasticity levels) due to the expansion of the cluster after the elasticity action is over (640 vs. 820 ops/sec). Performance under incremental streaming (Figure 2.3b) exhibits very little degradation into the early stages of elasticity (-2.5%) and a performance increase via the growing processing capacity of the joining node. This result highlights the benefit of decreased performance impact during elasticity actions with incremental streaming. Streaming takes 26 minutes for incremental vs. 9 minutes for parallel, with both systems taking additional time to reach their full post-elasticity throughput. Taking this additional time into account, we observe that Cassandra with incremental streaming takes about twice the time to reach its full post-elasticity throughput compared to parallel streaming. However, early benefits of the additional capacity of the new node are observed earlier (after 4 minutes) while the elasticity action.

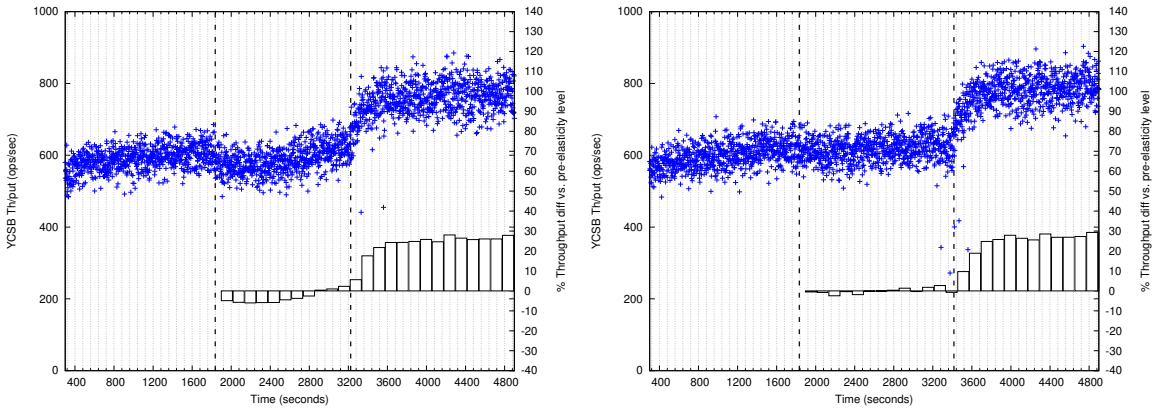
To statistically validate the results depicted in Figures 2.3a and 2.3b, we calculated the average performance change during streaming over ten runs. The maximum performance hit observed during parallel streaming over those runs was on average -13% (standard deviation 1.17%) with a maximum of -15.1%. During incremental streaming, the maximum performance hit over the ten runs was on average -2.3% at the beginning of streaming, with a near-linear improvement as the new node progressively joins the cluster.

One (existing) way to limit the performance impact of streaming is to actively throttle



(a) Parallel streaming (b) Incremental streaming

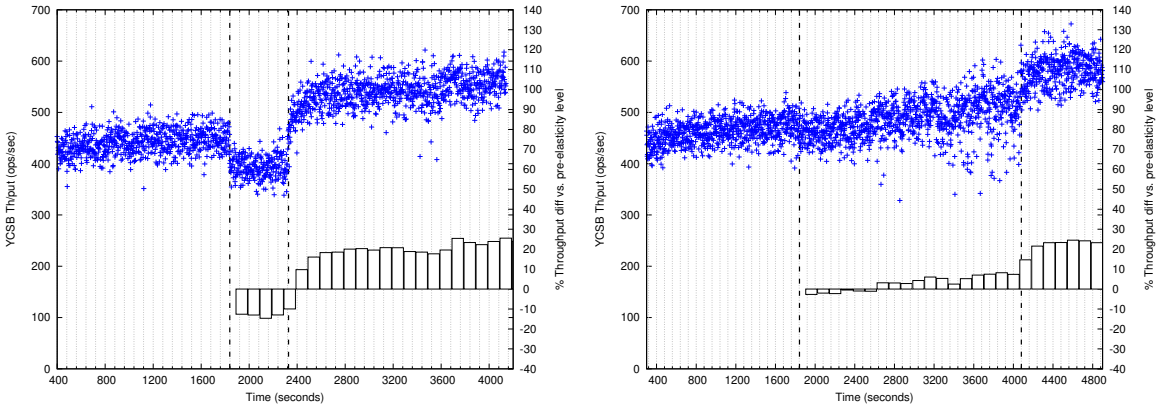
Figure 2.3: Elasticity under YCSB workload B (95% read, 5% writes), consistency QUORUM



(a) (b)

Figure 2.4: (a) Parallel streaming with network transfer throttling, (b) Serial *but not incremental* streaming. Both experiments run under the same configuration setting as in Figure 2.3

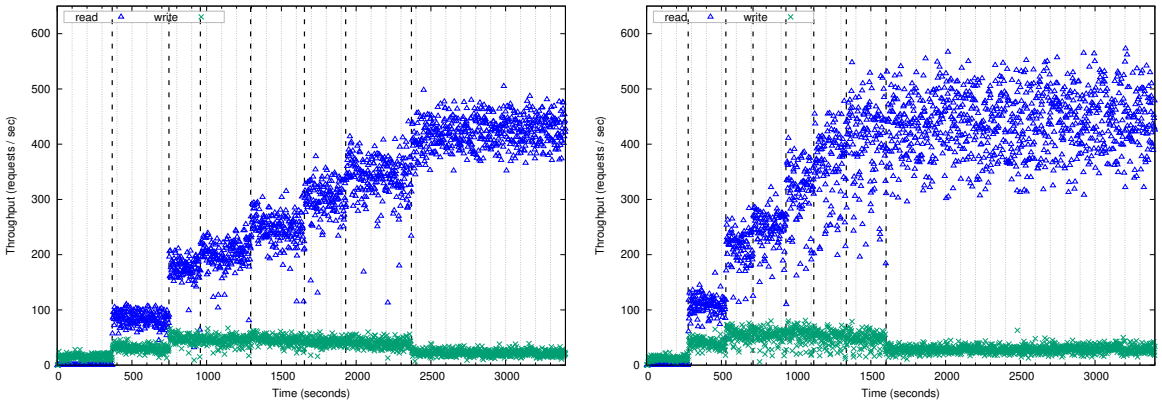
parallel streaming transfers through a Cassandra configuration setting (stream_throughput_outbound_megabits_per_sec). To highlight the benefits of incremental streaming over this solution, we repeat the previous experiment using standard Cassandra set to limit streaming throughput to a level comparable to the aggregate network throughput achieved with incremental streaming. Under incremental streaming (Figure 2.3b), the joining node receives tokens at 210Mbps (vs. 490Mbps with unthrottled parallel transfers). We thus set the throughput limit of each (parallel) sender to 30Mbps so that the joining node receives



(a) Parallel streaming

(b) Incremental streaming

Figure 2.5: Elasticity under YCSB workload B (95% read, 5% writes), consistency ALL



(a) Consistency Level ALL

(b) Consistency Level QUORUM

Figure 2.6: Request rate served by joining node during streaming, YCSB workload B (95% reads, 5% writes)

at the same rate (30 Mbps × 7 nodes = 210Mbps) as in incremental streaming. Figure 2.4a shows that the performance impact under throttled parallel transfers is reduced to -6% over a 24-minute elasticity interval vs. -14% over 9 minutes in unthrottled parallel streaming. Prolonging the elasticity action to reduce the performance impact results in late benefits of the elasticity action. So the system has to wait until the whole elasticity action is over before the capacity of the new node contributes to the overall system capacity. This also means that a large fraction of the processing capacity of the joining node goes unused, explaining the performance advantage of incremental elasticity.

Another way to showcase the positive impact of increasing the processing rate of the joining node during elasticity is to contrast it with a system that schedules sequential pairwise transfers (as in incremental streaming) but does not make transferred tokens incrementally available on the joining node. Results with such a system (Figure 2.4b) demonstrate that performance during elasticity remain at the same level as before elasticity, through the end of the streaming process: Although the joining node receives data from the existing servers in pair-wise manner (similar to incremental elasticity) the transferred data are not available for access to the new node. This means that the capacity of the cluster remains the same until the end of elasticity action when the new node announces the ownership of the received tokens and is fully integrated to the system (as in parallel streaming). Incremental elasticity schedules data transfers in pair-wise fashion in order to bring the additional capacity of the new node earlier into the system. As soon as a transfer is complete data become available for access on the new node during the elasticity action.

The increase of the cluster processing capacity in a stepwise fashion can also be observed by examining the rate of requests handled at the new (joining) server. Figure 2.6 illustrates that the joining node indeed serves progressively more client requests (reads/writes served locally –excluding those only coordinated[2] – by the joining node) during streaming. The time between two dashed lines in the graph corresponds to a streaming session between the joining node and an existing server. Each data point represents the rate of requests served in a time window of two seconds. Figure 2.6a shows the requests that are served by the joining node under consistency level ALL. In this case each request must be acknowledged by all replicas, thus the joining node is involved on all requests for tokens it has already received. Figure 2.6b depicts the number of client requests served by the joining node under consistency level QUORUM. In both cases, we can observe the increasing number of requests as the streaming process progresses. In case of consistency level QUORUM the steps are somewhat more noisy (the concentration of points is wider) as the new node may or may not be asked to participate in the majority (2 of 3) of replicas

[2]In Cassandra any read/write request can be sent to any node in the cluster. If the node is responsible for the requested key(s) it serves the request otherwise it acts as a proxy and forwards the request to the corresponding node. This proxy node is called *coordinator*. The coordinator is responsible for the entire request path and responds back to the client [9]

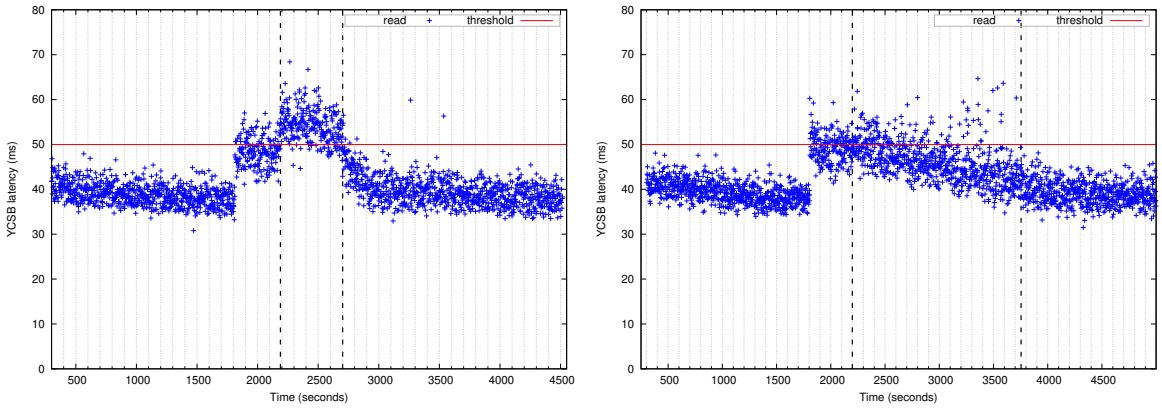
that serve a request. In Figure 2.6 we observe that the rate of write requests is higher during the elasticity action in both cases. This is due to the extra overhead of shadow writes during streaming.

Figure 2.5 depicts YCSB throughput for the same dataset for 95% reads 5% writes (workload B) under the stricter consistency level ALL. We observe similar trends as with the previous configuration applying consistency level QUORUM. With parallel streaming the throughput drops up to -17% during the elasticity action. Incremental streaming exhibits little degradation (-2.5%) at the start of the streaming phase. Under this configuration the performance increases in smaller steps during the elasticity action. Although with incremental elasticity one streaming node is active at a time, every request involves all replicas and thus performance is determined by the slowest replica. A small fraction of requests corresponding to tokens that are part of the active streaming session involve the streaming source node, and thus are expected to be impacted due to its higher load during streaming. Requests that do not hit this node will not be impacted under this consistency level.

Repeating these experiments with the 50%-50% read/write workload mix shows similar trends with the exception that the performance impact during streaming increases under both parallel and incremental streaming. With QUORUM consistency, parallel streaming exhibits steady performance degradation (similar to that observed Figure 2.3) up to -11%, while incremental elasticity exhibits a smoother transition to the new configuration with a maximum performance hit less than -4% during the early stages of elasticity, increasing to +10% as the streaming process progress. With consistency level ALL, parallel streaming exhibits a performance penalty up to -16%, whereas incremental elasticity exhibits a performance hit of up to -6.5% at the early stages of streaming, increasing slightly over the baseline (+3%) until the elasticity action completes.

2.3.3 Response time

The response-time trends observed in previous experiments are identical to those observed for throughput, with throughput drops corresponding to response-time increases. In this section we aim to extend our evaluation in two directions: First, to provide a response-



(a) Parallel streaming

(b) Incremental streaming

Figure 2.7: YCSB latency under workload B (95% read, 5% writes), consistency QUORUM

time view of incremental vs. parallel streaming. Second, to demonstrate a scenario of a variable-load client and a simple elasticity controller that drives an elasticity action to accommodate the increased load. Our evaluation sets the incremental elasticity technique into context with quality of service (QoS)-aware data stores that often rely on response-time targets to maintain specific goals [68, 86, 120, 140].

Variable levels of client load may occasionally result in a system exceeding a fixed response-time goal. A QoS-aware controller typically incorporates a monitoring component [68] that combines response-time metrics reported across client processes, analyzes, plans, and executes elasticity actions to re-provision service capacity as needed. We do not aim to fully evaluate an elasticity controller in this work. We use a simple such controller to compare the impact of incremental vs. parallel elasticity in maintaining a given response-time target. Figure 2.7 depicts YCSB read latency with a 95% read, 5% writes workload mix, consistency QUORUM, under variable load. The targeted response time of 50ms is highlighted with a horizontal red line in Figure 2.7.

Load generated by 20 YCSB threads results initially in 40ms average response time for read operations. 30 minutes into the experiment when the system has reached steady state, we increase the load by adding 10 more client threads to a total of 30, leading to an increase of the average response time above 50 ms. About 5 minutes later, the controller detects a response-time target violation and triggers an elasticity action whose duration is

highlighted by dashed vertical lines. Each data point in Figure 2.7 is average response-time over a 2-second window. Under parallel streaming (Figure 2.7a) we observe that the elasticity action leads to a further increase of response times, further exceeding our target. In contrast, incremental elasticity (Figure 2.7b) has lower impact on response time, producing fewer target violations and achieving a smoother transition to the new configuration.

To quantitatively compare incremental to parallel streaming with respect to violations of the response-time service-level objective (50ms in this case) that they produce, we count the proportion of those requests with response time exceeding our objective, using the following formula:

$$\text{score} = \sum_{i=S_1}^{S_n} \text{penalty}_i \tag{2.1}$$

where S_1 and S_n are the first and last interval of the streaming process, and

$$\text{penalty}_i = \begin{cases} 1, & \text{if resp. time} > 50 \\ 0, & \text{if resp. time} \leq 50 \end{cases} \tag{2.2}$$

The score function with the above penalty expresses the amount of time intervals that the elasticity process results in a target violation. The execution of Figure 2.7a (parallel streaming) results to a score of 237. This means that the response time exceeds the targeted threshold for 474 seconds (each time interval corresponds to 2 seconds), whereas the execution of Figure 2.7b (incremental streaming) has a score of 144 (144 x 2 = 288 sec), thus incremental streaming has 39% fewer response-time violations compared to parallel streaming.

In the above analysis, any violation of the response-time target is considered equally harmful and contributes a penalty of 1. However, one can argue that the amount by which the threshold is violated is also practically relevant. We can capture this factor by replacing formula (2.2) with (2.3) in the score function, thus taking into account by how many *ms* the

average response time exceeds the target at each data point.

$$\text{penalty}_i = \begin{cases} 50 - (\text{resp. time}) & \text{if resp. time} > 50 \\ 0 & \text{if resp. time} \leq 50 \end{cases} \tag{2.3}$$

According to this penalty function, parallel streaming has a score of -1170 while incremental streaming has a score of -440, indicating that incremental elasticity scores 2.6x higher than parallel streaming when taking into account both the number of response-time violations and their extent.

In Figure 2.7 we observe that Cassandra with parallel transfers is about twice as fast in bringing response time to its stable post-elasticity level compared to our implementation using incremental streaming. Thus if an application is insensitive to service-level violations or performance cost during elasticity, parallel streaming may be a preferable elasticity mechanism.

2.4 Related work

NoSQL systems, like nearly all data-intensive systems, require reconfiguration over time so that data is redistributed across server nodes for the purpose of load balancing, expanding/shrinking resources, or rehashing data to server nodes. Elasticity is a special form of reconfiguration where a fraction of the overall dataset is moved to new nodes (or taken out of nodes about to be decommissioned) to grow or shrink processing capacity in an online manner. In what follows we look into previous research on reconfiguration in NoSQL data stores, with a special focus on elasticity.

Several data stores are able to expand or shrink their use of server resources through migration of shards and replicas. Petal was an early elastic storage system offering a virtual disk abstraction featuring incremental reconfiguration [135], namely the ability to restripe a logical disk on fewer or more storage servers without blocking access to the entire disk. Incremental elasticity shares the concept of transferring a piece of the overall state at a time but has different goals.

A number of NoSQL data stores (sych as Dynamo [86], Cassandra [132], Riak [5], and

Voldemort [175]) use a variant of *consistent hashing* originally introduced by Karger *et al.* [119] and Stoica *et al.* [171], to map key ranges to server nodes. Instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring, each such point called a *virtual node*. Each node in the NoSQL system can be responsible for more than one virtual node. Systems using virtual nodes involve all-to-one transfers during elasticity and can benefit from the use of incremental elasticity.

Ghosh *et al.* [100] described Morphus, a methodology for supporting online reconfigurations in sharded NoSQL systems. Morphus introduces algorithms for re-sharding a database aiming to reduce the total network transfer volume during reconfiguration and to achieve a load balanced assignment. Morphus is implemented within MongoDB and applied to the case of changing the sharding key of an existing table. It leverages internal MongoDB mechanisms to reconfigure secondary shard replicas and transfer data via many-to-all communication patterns inherent in such resharding scenarios. Changing the sharding key entails a heavier reconfiguration compared to adding new servers considered in our work.

DDS [104] was an early scalable key-value store with the ability to split partitions (shards) and migrate replicas between nodes. It chooses shard size so that migration of a shard is a rapid process and opts for migrating replicas at the full speed of the network rather than in a controlled, background process. The authors of DDS do not detail a complete elasticity mechanism.

The term *incremental elasticity* has also been used in the context of array databases [90]. In this work, Duggan *et. al.* design an elasticity controller that decides when to expand an array database cluster and how to repartition a growing multi-dimensional data set within it. Our use of the term refers to progressive increases of processing capacity in a fine-grain manner during an elasticity action and is thus complementary.

MongoDB utilizes the cluster balancer module, which migrates data chunks between different shards when the chunk number ratio of the biggest shard to the smallest one reaches a certain threshold. MongoDB appears to support an elasticity mode where data is being served by newly added nodes as soon as they arrive [89]. Based on the scarce documentation available for this feature, it does not appear related in any other way to

incremental elasticity. Voldemort [175] uses a rebalance controller that allows a joining node to participate in multiple parallel transfers, ensuring that each sender has only one active token-transfer at a time. Riak [5] supports the adjustment of the number of node-to-node transfers using a per-node *transfer_limit* parameter (default 2). It appears that this parameter controls the number of tokens involved in data transfers rather than the number of parallel transfers towards a joining node. It has been used on disk-capacity issues in expanding clusters [35] but we have found no systematic evaluation of its performance impact or a specification of how data transfers using it are coordinated/managed across nodes.

Konstantinou *et al.* [127] describe a generic distributed module, DBalancer, that can be installed on top of a typical NoSQL data-store and provide an efficient configurable load balancing mechanism. Balancing is performed by simple message exchanges and typical data movement operations supported by most modern NoSQL data-stores. In a related work, Konstantinou *et al.* [182] present a cloud-enabled framework for monitoring and adaptively resizing NoSQL clusters. Kuhlenkamp *et al.* [130] evaluate the elasticity of HBase and Cassandra and show a tradeoff between the speed of scaling and the performance variability while scaling. These works rely on the use of generally available elasticity mechanisms and thus could benefit from the use of the *incremental elasticity* mechanism.

Data stores offering performance-oriented service-level agreements (SLAs) such as Dynamo [86] use online elasticity mechanisms as one among different ways of adjusting processing capacity to fit client needs. Typically, a combination of vertical (increasing single-node processing capacity) and horizontal (increasing number of nodes) elasticity is used in this space, also known as scale-up and scale-out elasticity. Incremental elasticity is an instance of the latter. Recent work [68] proposed a measurement-based prediction approach [121] to achieving data store performance SLA by means of elasticity actions over the Cassandra NoSQL store. Here we improve on this work by reducing the impact of elasticity actions.

Brown *et al.* [64] introduce a methodology for benchmarking the availability of RAID arrays after a disk crash, highlighting a trade-off between the speed of data reconstruction and its impact on application performance, similar in spirit to the elasticity tradeoff

explored in our work. They study the policies used by different software RAID implementations (Solaris, Linux, Windows) and find that Linux follows the slowest approach, dedicating less disk bandwidth to reconstruct data posing no significant effect on application performance, whereas Solaris defines the opposite extreme making its RAID reconstruction over 7 times faster than Linux, albeit at a significant performance hit during reconstruction.

2.5 Summary

In this chapter we describe incremental elasticity as a way to reduce the performance penalty incurred during elasticity actions in distributed data stores. Our implementation on the Cassandra column-oriented data store shows that such an implementation is feasible with reasonable complexity. Our evaluation shows that incremental elasticity results in smoother, more stable elasticity actions compared to parallel network transfers across all cases considered. Incremental elasticity reduces the performance impact (-2.32% vs. -12.96% drop in aggregate throughput for 95%-5% reads/writes and QUORUM consistency, -2.5% vs. -17% under 95%-5% reads/writes and ALL consistency) compared to parallel network transfers. In a scenario involving a service-level management controller, incremental elasticity leads to 39% fewer response-time violations compared to parallel streaming during elasticity.

Chapter 3

Replica-group leadership change as a performance enhancing mechanism

During the lifecycle of a data store the system faces a number of challenges as it evolves through multiple phases. The challenges a system faces can be internal or external. Internal data reorganization or backup and checkpointing background tasks competing for resources can be the source of performance variability resulting in violations of its performance oriented goals. Data stores maintain, organize and store data using data structures. There are various types of data structures for storing data offering different and unique features each one [113]. During its lifecycle, a data store performs data reorganization operations on its internal data structures allowing continuous and efficient operation. Log structured merge tree (LSM tree) [146] is a popular data structure underlying many highly scalable distributed data stores [36,49,71,86,98,132] and embedded key value stores [10, 16]. Periodic compaction operations (merge sort operation over the data) are necessary actions to reduce space amplification and improve read operations over time. However, the compaction tasks lead to a lot of overhead, such as CPU resources consumed by data compression, decompression, copying and comparison, and also the disk I/O of data reads and writes. Data backup is a data protection method that creates a copy of the store's state allowing it to restore the state in case of a failure or roll back if the system's state is corrupted. Regular backups keep the data stores state safe and sound. However,

backup operations put additional strain to the system as it creates significant I/O activity, memory pressure and CPU overhead. Thus, the system may fail to sustain the expected performance while such background activities are active.

Data replication is a standard technique for achieving high data availability and reliability, and a range of data replication techniques are essential components of distributed storage systems today [72]. Dynamic reconfiguration of replica groups has received significant attention recently [62, 137, 147, 168] as the importance of adjusting the number and type of nodes backing replica groups with minimal downtime has become a key system requirement of Internet applications.

In this chapter we aim to systematically explore replica group reconfiguration policies and mechanisms as a means of adaptation action that can mask the performance bottlenecks imposed by data store's internal and external background tasks. We describe the general design of such a system, evaluate multiple sources of overhead (LSM-tree [146] leveled compactions and data-backup tasks) over a wider range of workload mixes in a dedicated cluster, using an industrial-strength implementation based on MongoDB and an LSM-tree based storage manager, RocksDB [10].

We outline the general design of such a system by defining the states each replica can be in (primary or secondary, performing a background task or not) and their transitions, and highlight the key policies and mechanisms involved. We further present key implementation choices we made within MongoDB/RocksDB, in maintaining global information about node activities for ranking nodes as candidates for primary, coordinating between the independent replication (MongoDB) and storage (RocksDB) layers, and ensuring that our preferred candidate can always be elected by the PB election algorithm.

Our experimental evaluation highlights the performance benefits of reconfiguration actions in the case of LSM-tree compactions and data backup processes. There are other known performance issues, such as slow system response during checkpoint writing [24], that could be similarly addressed by our system and that we plan to evaluate in future work.

Our key contributions in this chapter are:

- A description of the design concepts (replica states, transition policies, and under-

lying mechanisms) in using replica-group leadership change as a performance enhancing mechanism in primary-backup replication

- Addressing key challenges in an implementation of the design within an industrial-strength NoSQL data store (MongoDB) and storage manager (RocksDB) using LSM-trees with leveled compaction
- An evaluation of the prototype under two sources of periodic performance impact on replica-group nodes: LSM-tree compactions and data backup tasks.

The remainder of this chapter proceeds as follows: In Section 3.1 we provide background on the primary-backup (PB) replication scheme as well as details on the MongoDB replication internals and RocksDB storage engine based on LSM trees. In Section 3.2 we describe the design and implementation of our system. In Section 3.3 we discuss our experimental evaluation, and in Section 3.4 we conclude.

3.1 Background

As background to our design and implementation in Section 3.2 we describe the basic operation of replication on MongoDB [23] and RocksDB [10], the storage manager we use in this work. In general, MongoDB stores data in documents. It groups documents in *collections* (akin to tables), partitions data via *sharding*, and replicates shards as explained below.

Primary-backup replication. Data replication schemes are prevalent in data stores for high availability and reliability as data are replicated across a set of nodes, comprising a replica group. A range of existing replication techniques in distributed storage systems dictate how data are propagated to replicas.

Primary-backup (PB) replication [65] is a strongly consistent replication technique in widespread use today [117, 145, 147, 187]. Systems implementing PB replication feature a strong leader (the primary) that coordinates write operations (in some systems, reads as well) towards secondary replicas (backups) (Figure 3.1). Any PB implementation must

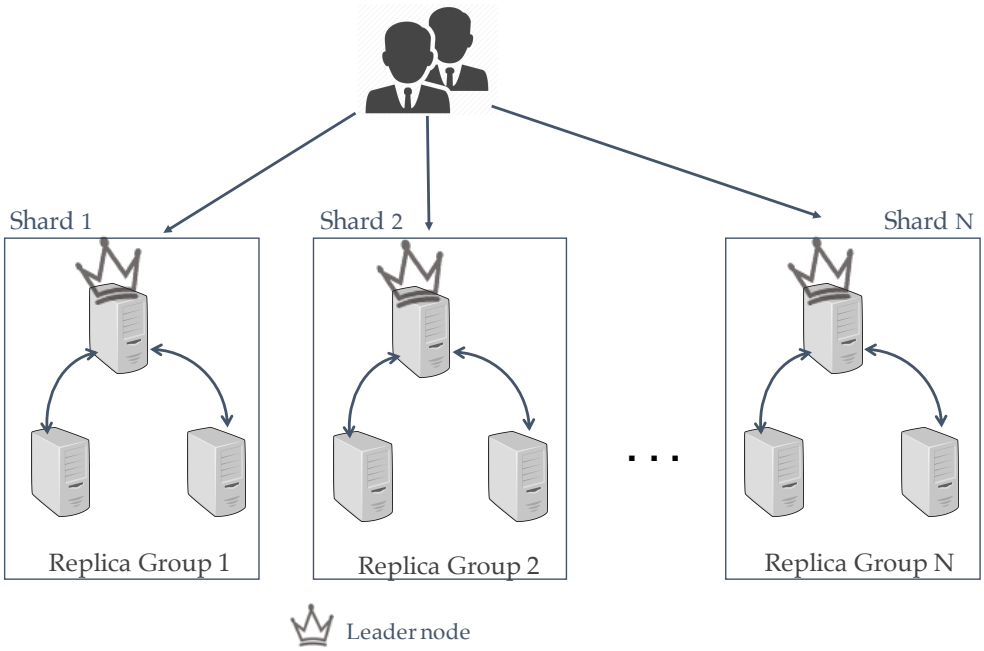


Figure 3.1: The primary-backup (PB) replication scheme. Each data partition (shard) comprises a replica-group featuring a strong leader (primary) node that coordinates the client requests and a number of replicas

handle failure of the primary by electing a new primary within the current configuration, as a standard reconfiguration action (a *view change*). In recent years, certain PB implementations offer APIs that externally trigger such reconfiguration actions for management purposes.

Being on the critical path of I/O activity, the primary in a PB system typically takes higher load than secondary replicas. Any additional overhead on the primary may critically affect performance of the replica group as a whole. Recent research [96] demonstrated use of targeted leadership-change actions as a driver for lightweight adaptation of replica groups, by moving the primary away from nodes that are, or will soon be, heavily loaded. In this way, a certain level of load balancing is feasible at low cost (the short availability lapse that such reconfigurations entail), occasionally leading to large benefits. Leadership change has also been proposed in the context of Byzantine fault tolerance for mitigating attacks in which a leader intentionally misbehaves [79]. Use of leadership-

change in BFT settings, while partly sharing goals with our work, differs in terms of the policies and mechanisms involved.

MongoDB replication. MongoDB replicates shard data using primary-backup replication [65]. Each replica group has a single primary and multiple secondaries. Our brief discussion of the replication protocol is based on online documentation [26] and inspection of the source code (MongoDB version 3.7). A primary is associated with a given *term*, indicating the election number at which it was elected. The primary inserts each client update it receives to an operation log (the *OpLog*) stored as a local file and then (asynchronously) applies it to its local copy of the database. Secondaries pull OpLog entries from the primary or from another secondary, known as their *sync-source* node. Each secondary stores fetched updates to its own OpLog and applies them to their own database copy. In our experiments the sync-source is always set to be the primary.

All MongoDB nodes maintain topology and status information about other nodes in the cluster. Each node communicates regularly (in heartbeats every 2 seconds) with all other nodes to check their status, to stay up to date with their sync source (fetching OpLog entries), and to notify them of their progress.

A node runs an election when it has not seen a primary within the election timeout, or during explicit reconfiguration (termed a *priority takeover*). The election process is based on the Raft [147] protocol. When a candidate wins an election, it notifies all nodes via a round of heartbeats. It then checks if it needs to catch up with the former primary. This process tries to commit as many as possible of the entries the last primary managed to send to secondaries, but may not have managed to commit (they will otherwise be rolled back). Prior to accepting writes, the primary-elect drains its OpLog from previous-term entries by applying them to the database.

RocksDB. The implementation of MongoDB we employ in this work uses RocksDB [10] as a storage engine. RocksDB stores data in Log-Structured Merge (LSM) trees [146]. It applies each incoming update to a write-ahead log for durability and then to an ordered memory buffer (the memtable), periodically flushed to an immutable indexed ordered

file (the SSTable). To retrieve a requested key, a number of SSTables may have to be consulted. Periodic compactions (merge-sort runs) of SSTables aim to maintain a small number of large SSTables, improving read performance. RocksDB implements leveled compactions [22], initially proposed in LevelDB [16], a key-value storage engine written at Google.

The MongoDB OpLog is implemented as a specific collection type called a *capped collection*, built as a circular buffer. It is a fixed-sized collection that automatically overwrites its oldest entries when it reaches its maximum size. MongoRocks monitors the size of the OpLog collection and reduces it via compaction. Due to heavy overwrite activity, the OpLog involves significant data-discarding during compactions, affecting overall performance. Thus MongoRocks triggers compaction periodically even if the size of the OpLog has not reached its limit. OpLog compactions have higher priority in the queue of pending compaction operations.

3.2 Design and Implementation

Design. Our design goal is to mask the impact of resource-intensive background activities on replica-group performance. Such activities, caused by internal storage-system needs (such as LSM-tree compactions) or external tasks (such as automated backup jobs), are often essential for smooth operation and cannot be avoided or postponed for too long, as doing so impacts application performance and/or data availability. The key idea is to demote a primary who is about to engage in such a resource-intensive background task into a secondary, in effect executing such tasks on secondary nodes only. The key mechanisms that facilitate this design are:

- Notifying the primary that a background task is upcoming
- Reconfiguring the group with minimal service disruption.
- Maintaining a global view of background tasks executing on replica group members. Each replica n_i periodically gossips the following status to the group: whether there is a background task (internal or external) executing on n_i; when the last background

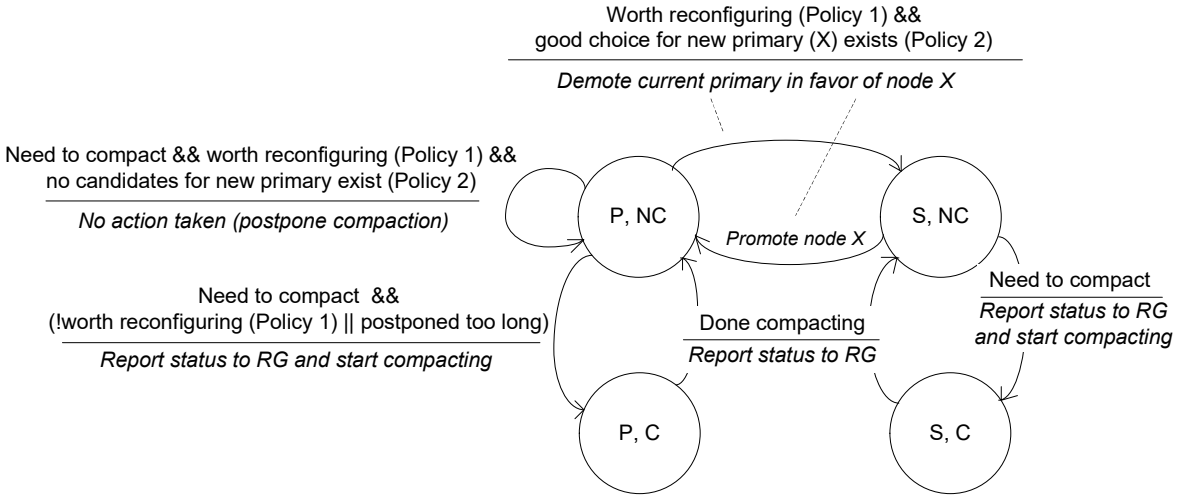


Figure 3.2: States of a single replica (P: primary; S: Secondary; NC: not compacting; C: compacting) and transitions

activity completed on n_i.

A state machine describing the states a replica can be in and possible transitions is depicted in Figure 3.2. For concreteness and without loss of generality we assume that the background activity we aim to mask is LSM-tree compactions, however other (and different types of) activities are supported. A primary starts in state P, NC (primary, non-compacting) and each secondary in S, NC (secondary, non-compacting). A reconfiguration moves the primary from P, NC → S, NC and a secondary from S, NC → P, NC. Two key policy decisions are when to trigger a reconfiguration and when that happens, which secondary is the best choice for new primary:

Policy 1: When is reconfiguration beneficial. A primary decides that a reconfiguration of the replica group is worthwhile when a background task is upcoming on the primary and the anticipated performance impact justifies the cost (short availability lapse) of reconfiguration. The impact of the type of background tasks we consider in this work (LSM tree compactions, data backups) nearly always justifies a reconfiguration. In future work we intend to broaden our investigation to a wider range of background activities, including shorter background activity spikes.

Policy 2: What is the best choice for new primary (if any). Another key policy is se-

lecting the replica to promote to next primary. The selection process is based on a *replica-ranking* (RR) algorithm that takes into account the gossiped state of all replicas. The RR algorithm on the primary considers all secondaries that are not executing an internal or external background task as candidates for new primary. If there are more than one candidates, the algorithm by default favors the node that most recently completed an internal background task (*"most recently completed"*, MRC). Since internal tasks (such as compactions) are usually periodic, the expectation is that this candidate should enjoy a longer background-activity-free period until its next internal background task.

Another option is to select the candidate that served as primary for the longest time in the past (*"longest-served primary"*, LSP). Since all replicas maintain an in-memory read cache but only nodes that serve client requests (primaries) use it (secondaries only fetch updates to their OpLogs, bypassing their cache), LSP effectively favors nodes with warmer caches.

Policies 1 and 2 (*when to reconfigure* and *whom to promote*) may in some cases be interrelated: A reconfiguration may be called as soon as the background task on a recently-demoted primary is over (even though there may not be a pending compaction on the current primary), turning the leadership to the previous primary. This joint policy choice (an LSP variant we term *"preferred primary"*) in which a single node acts as primary, except for periods when it performs heavy background tasks, is geared to promote high cache hit rates.

If there is no suitable candidate (e.g., if all secondaries are compacting), the primary postpones its background activity (P, NC → P, NC) and re-evaluates its decision as soon as it receives a status update from any of the secondaries that may point to a suitable candidate. The primary decides to start the activity anyway (P, NC → P, C) if the reconfiguration is not worthwhile or if it has been postponing it for too long. Secondaries that need to perform a resource-intensive background task are allowed to proceed (S, NC → S, C). During that period they are ineligible to become primary.

A key challenge for our work is that a suitable candidate for primary may not be always easy to elect in existing variants of primary-backup [26, 117, 145, 147]: Most such election algorithms require that the elected primary is a node that has seen all committed propos-

als, whereas our optimal choice for next primary may be a node that has not participated in a committing majority for a long time (e.g., due to a heavy activity in the past on their side). To ensure that our choice for new primary does not lead into a state where the node cannot be elected within a reasonable amount of time, if the candidate fails to get a majority of votes (voters' OpLog being ahead of the candidate's OpLog) the current primary steps down (prohibiting new updates) so that the primary-elect catches up with the latest OpLog entries and request that replicas vote again, eventually ensuring success.

Implementation. Our implementation is based on MongoDB [23] with the RocksDB engine [10] (MongoRocks [28]).

MongoDB has built-in support for reconfiguring a replica group. Replicas regularly exchange a ReplicaSetConfig structure, listing all nodes in the replica group at a particular configuration *version*. Changes in the configuration are indicated by a change in the version number and propagated from the primary (downstream) to all nodes. Each replica-group member has a *priority* property that affects the timing and thus the outcome of elections for primary. When a node learns of the new ReplicaSetConfig, it ranks all of the priorities listed there and assigns itself a timeout proportional to its rank

$$(\text{priority rank} + 1) \times \text{election timeout}$$

After the timeout expires, it checks if it is eligible to run for election, and if so it starts an election.

In our implementation, we programmatically assign priorities to each node in a new ReplicaSetConfig as indicated by our replica-ranking (RR) algorithm. The node we aim to elect as the new primary has higher priority than the current primary. We modified the priority takeover timeout to activate the action as soon as possible. In practice, reconfiguration completes within a few hundreds of milliseconds.

Each replica-group member maintains a view of the current background tasks running on all replicas as input to the RR algorithm. All members periodically report their status over the regularly-exchanged MongoDB heartbeats. We use the RocksDB internal *com-*

paction statistics API to expose the counter of currently-running compactions (*rocksdb.num-running-compactions* property) to the MongoDB layer and embed this information to every heartbeat message. The receiver extracts information from heartbeats from each replica-group member and derives when the last compaction ran on each node. Our RR algorithm selects only between non-compacting nodes and by default favors the one with the most-recently completed compaction (*MRC* policy). In addition to LSM-tree compactions, the current implementation carries information about and handles externally-triggered data backup tasks.

Reconfigurations of the replica group can only be triggered by the primary. To reinstate the previous primary (*"preferred primary"* policy), the current primary monitors the status of its predecessor (now a secondary) based on its periodically reported status. When the current primary notices that the "preferred primary" has completed its internal background activities, it triggers elections proposing it as a candidate. We have observed in our experimental evaluation that the preferred-primary policy promotes good cache behavior as the primary maintains a fresh cache for a long period of time.

RocksDB communicates with MongoDB over a sockets channel to notify it of a compaction task and ask its permission to start it. If the MongoDB replica is a secondary, it responds positively. If it is primary, MongoDB asks RocksDB to defer the compaction and starts a reconfiguration. RocksDB uses a *pending-compaction-queue* to remember the database that requested the compaction. When the node transitions to secondary, RocksDB goes over entries in the *pending-compaction-queue* to reconsider delayed compactions. We use MongoDB's *ReplicationCoordinator* public API to expose information about replica state transitions to the storage engine.

MongoRocks uses compaction tasks to reduce the size of the OpLog. By default it triggers OpLog compactions at least every 30 minutes, even if the OpLog has not reached its limit. Assuming that all members of a replica set start at the same time, this would normally lead all nodes to perform their OpLog compactions in sync. In this way there would be no available non-compacting replica to replace the primary. In our implementation we randomize this time interval on every node to be between 30 to 45 minutes. In our experiments the system was always able to find a non-compacting replica.

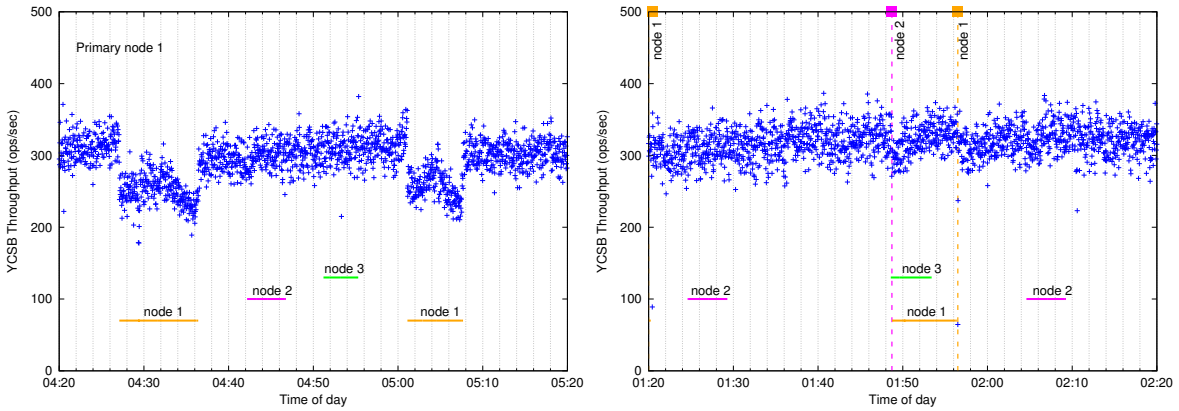
3.3 Evaluation

Our experimental testbed is a cluster of 4 servers, each equipped with a dual-core AMD Opteron 275 processor clocked at 2.2GHz with 12GB of main memory. All servers run Ubuntu 14.04 64-bit with a 3.14.1 Linux kernel and are interconnected via a 1Gb/s Ethernet switch. Servers used as MongoDB servers have a base 72GB 10,000 RPM SCSI drive with an additional 300GB 15,500 RPM SAS drive dedicated to storing data. All hard drives are formatted with ext4.

We use MongoDB 3.7 with RocksDB 5.7 as storage engine. Our evaluation workload is the Yahoo Cloud Serving Benchmark (YCSB) [81] version 0.11 executing on a dedicated server. The benchmark is configured to produce two different mixes of reads vs. updates/writes: 90%-5% (read intensive) and 50%-50% (write intensive), with requested keys selected randomly with the Zipf distribution. The YCSB evaluation dataset consists of 12 million unique records or 15GB of data per node. Our initial MongoDB cluster consists of one shard and 3 servers in the replica group. The number of YCSB client threads (number of parallel connections between database client and servers) is set to 2. We set MongoDB's *write concern* to one, meaning that writes are considered complete when acknowledged by the primary. We use journaling, which requires OpLog entries to be durable (written to disk) before acknowledging. We use the default settings for RocksDB: *max_background_jobs*, the maximum number of concurrent background jobs (including flushes and compactions) is set to 2, and *compaction_style* set to *level*. The dataset is loaded fresh onto MongoDB nodes before each experiment. In the following two sections we evaluate the benefit of reconfiguration actions in the presence of LSM-tree compactions and data backups. The default RR policy, MRC, is used in all experiments.

3.3.1 LSM-tree compactions

Figure 3.3 depicts YCSB throughput (ops/sec) under the read-intensive workload mix. As in all subsequent figures, time (x-axis) starts one hour into the experiment, when the system is deemed to have reached a steady state. Figures 3.3a and 3.3b depict performance without and with reconfigurations. Colored squares at the top of a graph (and correspond-



(a) Standard MongoRocks (primary node: 1)

(b) MongoRocks with reconfigurations

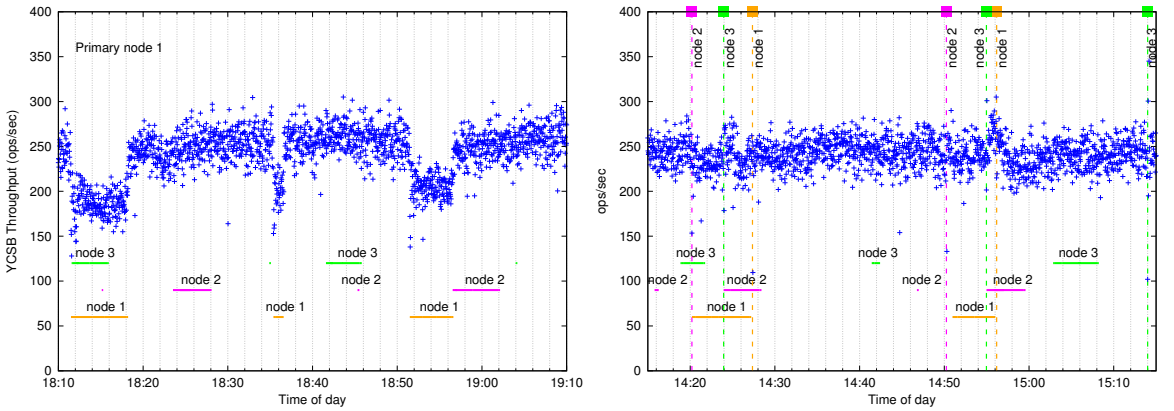
Figure 3.3: 90% reads-10% writes

ing vertical dashed lines) represent elections of a node as primary. Horizontal solid lines represent compaction tasks performed on a node. Each line may represent more than one sequentially-executing compaction tasks at different levels [22].

Figure 3.3a exhibits a clear performance hit during the compaction tasks at the primary node (indicated by horizontal orange lines labeled “node 1”). The average throughput during the one-hour window shown is 295 ops/sec. During primary compactions the average throughput is 253 ops/sec, whereas non-compacting time periods have an average throughput of 310 ops/sec. Thus there is a 18.4% performance hit during compaction tasks at the primary.

Performance exhibits a more stable behavior under the automated replica-group leadership change mechanism (Figure 3.3b), achieving an average of 318 ops/sec over the one-hour window. Overall our system exhibits 7.2% higher throughput during the one-hour window. Compaction tasks in this experiment were mostly due to growth of the OpLog database.

Figure 3.4 presents YCSB throughput under the write-intensive workload. Figure 3.4a exhibits a clear performance hit during the compaction tasks on node 1 (primary). The average throughput for the one-hour window is 239 ops/sec. Average throughput during compactions is 194 ops/sec, whereas non-compacting periods (on primary) have an average throughput of 252 ops/sec. There is thus a 23% performance hit during com-



(a) Standard MongoRocks (primary node: 1)

(b) MongoRocks with reconfigurations

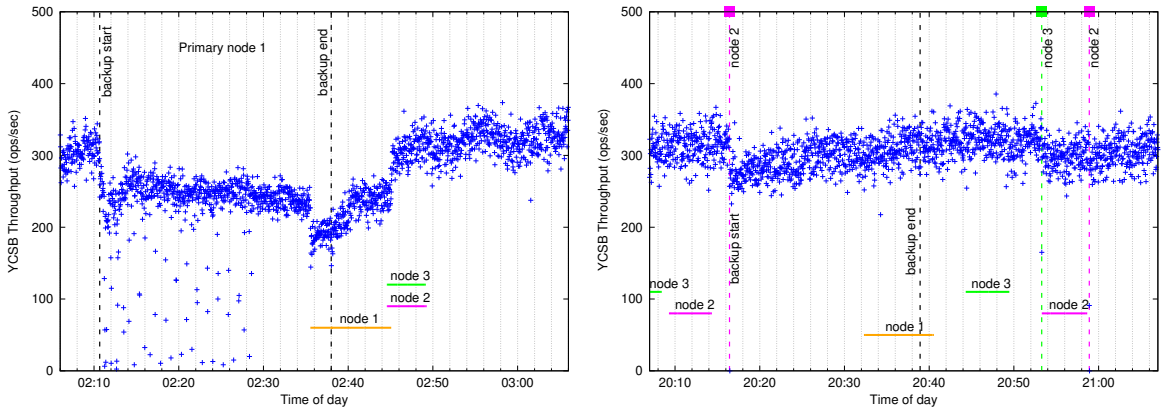
Figure 3.4: 50% reads-50% writes

pactions on the primary. Throughput using reconfigurations exhibits a more stable rate (Figure 3.4b) achieving on average 250 ops/sec. Under the write-intensive workload the system achieves on average 4.2% higher throughput. We observe that our results are not sensitive to the read/write ratio as compactions are mostly OpLog-driven and in both cases have relatively low frequency. However, the performance improvement of reconfigurations in these representative runs is significant (18%-23%) during compactions in both cases.

3.3.2 Data backup

We extend our evaluation to external background tasks, and study performance while taking an online data backup with the MongoDB mongodump utility [27]. We use the *–oplog* option to capture incoming writes that occur during the mongodump operation into a file, ensuring that backups are point-in-time snapshots of the database state. During the backup process, the tools force a running database instance to read all data through memory. Reading infrequently-used data has the side effect of evicting frequently-accessed data from the cache, causing the performance of the regular database workload to deteriorate.

Figure 3.5 exhibits the impact of the backup process on performance while executing a read-intensive workload. Figure 3.5a shows a clear performance hit during the backup



(a) Standard MongoRocks (b) MongoRocks with reconfigurations

Figure 3.5: Data backup (mongodump –oplog), 90% reads-10% writes

task (whose duration is indicated by the vertical dashed lines) when there is no reconfiguration action. The average throughput achieved during the one-hour window is 266 ops/sec. At 02:30 mongodump completes the extraction of data and begins to copy OpLog entries. The OpLog dump has no effect on the cache in contrast to the previous phase, where throughput was more noisy. At the end of the backup process there is an overlapping compaction task that results in a further performance degradation. The average throughput during the background tasks (backup and compaction) is 229 ops/sec, whereas average throughput when no background task executes is 314 ops/sec. The system achieves 35% higher throughput when there are no background activities on the primary.

Figure 3.5b exhibits performance when applying a reconfiguration action prior to the backup. At 20:16 the system is notified about the upcoming external background task and the primary initiates a replica-group reconfiguration to mask its impact. The RR algorithm selects node 2, the replica with the most recently completed compaction (default policy), as the new primary. Node 2 serves as primary for the first time and therefore it initially suffers from low cache-hit ratio (causing the performance dip at 20:16), gradually increasing performance. This has not been a problem in previous experiments as the role of primary had been rotated around nodes a few times already before the measurement phase began, warming up caches. In this case, the average throughput during the one-hour window is 306 ops/sec, 15.8% higher average throughput than standard (non-reconfiguring)

MongoDB. In addition, our system benefits the data backup task itself, as there is lower contention for resources in secondary nodes. The elapsed time of the backup task in our system is 21.5% shorter than the non-reconfiguring system (1,349 vs. 1,640 sec).

3.4 Summary

In this chapter we evaluate the performance benefits of automated replica-group reconfigurations under the impact of periodic sources of overheads such as LSM-tree compactions and data backup tasks, using the YCSB benchmark. We find that targeted leadership change actions lead to 18-23% improved performance during execution of compaction tasks and 4-7% improved performance overall for LSM-tree compactions. Replica-group leadership change prior to a data backup task on the primary leads to 35% better performance and 21.5% faster backup compared to standard MongoRocks.

Chapter 4

Addressing the read-performance impact of replica-group reconfigurations

Replica-group reconfiguration actions occur for many reasons such as performance enhancements (as we described in Chapter 3) but also due to failures, load balancing and workload migration. Nevertheless we observe a performance glitch within the reconfiguration action. In this chapter we study the performance impact during the replica-group reconfiguration actions. Based on our observation that the serving nodes suffer from high cache-miss rate during the transition, we propose a method to eliminate the cold-cache effect.

Raw data is often several times larger than available memory, and performance of storage devices is still orders of magnitude lower than that of main memory. In combination with today's read-dominated production workloads and strong temporal locality among requested keys [57], application performance still depends on the efficiency of data caching. Not surprisingly, modern data stores maintain in-memory data caches to improve performance, especially for read-dominated workloads. The efficiency of a read cache impacts the overall performance of data intensive systems.

In replicated data stores, data is replicated across a set of nodes comprising a *replica*

group. A range of existing replication techniques in distributed storage systems dictate how data are propagated to replicas. Based on the consistency model used, replica groups ensure that replicas within them apply updates in some specific order (strongly consistent systems require that replicas totally agree on this order). Data replication techniques focus on ensuring consistency of the persisted data set, while each node of the replica group also maintains a memory cache of that set to improve performance.

There are multiple replication models in use today. *Primary-backup* (PB) replication [65] is a traditional replication technique in widespread use [117, 145, 148, 187]. Systems implementing PB replication feature a strong leader (or primary) that coordinates read and write operations towards secondary replicas (or backups). Reads are typically served by a single replica, most commonly the primary itself. Many such systems allow clients to read from any single replica to better distribute read-intensive workloads. *Active replication* systems, such as those based on the Paxos algorithm [133], have also been in use in storage systems [63, 69]. Formally such systems involve several replicas to carry out operations, typically a majority. Efficiency considerations in today's read-dominated workloads [57] have led to active-replication systems that permit reads from a single replica, similar to PB systems, through *lease-based* mechanisms [103, 134, 142]. *Quorum-based* systems is another class of replication systems where reads involve a set of nodes [58,86,101]. Efficiency considerations in quorum-based systems have also led to configurations featuring a small read set (often a single replica [116]). We can thus identify a trend in practical systems to restrict reads to as few replicas as possible, to optimize for read-dominated workloads.

Involving as few replicas as possible in read operations means that the caches of the remaining replicas receive delayed or no information about application reads, and thus do not fetch up-to-date content in memory, in contrast to replicas serving reads. This often does not pose a problem, when reads are satisfied consistently by a single or the same set of replicas. However, when the replica(s) satisfying reads shift to replicas that have not actively maintained their read cache, a performance impact is incurred for read operations.

A switch of the read-serving replica may occur as a result of different types of *reconfiguration* operations. One such case is the frequent (e.g. every 20 minutes) replica group

reconfiguration include lightweight adaptation actions as a performance enhancement mechanism to mask performance bottlenecks on the serving nodes (Chapter 3). Replica-group reconfiguration traditionally occurs when a serving replica fails and a backup must take over (failover) or due to load balancing actions or workload migration where parts of an application are moved across the infrastructure. In a similar case, geo-replicated systems reconfigure the set of serving replicas (e.g. leader leader) while the client's location shift across time zones as they serve Internet users [53, 138, 166, 189]. While such reconfiguration actions have a positive long term impact, a negative side effect at the time of switching the read-serving node, is that a burst of cold-cache misses at the new read-serving replica may lead to latency spikes and throughput drops that result in service-level objective (SLO) violations for significant amounts of time (several minutes).

In this chapter we address this problem by maintaining up-to-date read caches across all replicas *without* increasing the number of replicas actually serving reads or modifying the replication semantics of the system. We achieve this through a low-overhead mechanism by which all replica nodes eventually see a tunable subset of read operations (loading the respective data into their in-memory data cache) in the order seen by replicas serving reads and primarily responsible for serving client requests. This method eliminates the cold-cache effect observed after a reconfiguration on the new node that serves reads, which impacts the overall system performance seen by clients. Our proposed solution is a lightweight, best-effort synchronization method that tries to minimize the overall cache divergence between the primary and replica nodes, by disseminating read requests across replicas in the background (not tied to the execution of client requests). To achieve this, read serving nodes keep a volatile buffer containing information about the reads executed by the replica group. The buffer is periodically disseminated to replicas not involved in serving reads and is used as hints to update their caches. The design of this mechanism is based on basic principles and can be easily integrated to production replicated key-value stores such as the widely used MongoDB. Our evaluation demonstrates that the overhead is minimal and that the prototype maintains stable performance eliminating SLO violations during reconfiguration actions.

The rest of the chapter is organized as follows. Section 4.1 describes background and

related work and Section 4.2 describes our design choices and implementation details. In Section 4.3 we present evaluation results, and in Section 4.4 we conclude.

4.1 Background and Related Work

Two major points relating to our work are (i) replication systems today read from a subset of replicas (typically one), leading to situations where backup (non-reader) replicas have an out-of-date memory cache; and (ii) under certain situations (such as after a failure of the primary or other reconfiguration action) reads are directed to such replicas, leading to a surge in cache misses. In what follows we will describe how different replication systems relate to points (i) and (ii). There is currently no system that directly addresses this problem (short of issuing reads to all replicas, which penalizes the common path of read operations and thus avoided in practice).

Modern systems often read from one replica per group (shard). The design and implementation details of replication mechanisms (such as, for instance, how read and write operations are performed) vary across systems, entailing consistency, availability, and performance tradeoffs [102]. A trend in practical replication systems is to restrict reads to often just one (dynamically selected) replica per shard, to optimize for read-dominated workloads.

In quorum-based [86,101] or active-replication (Paxos) [63,69] systems, read and write operations are typically issued towards sets of replicas (or *quorums*), which can generally differ for reads and writes, and whose sizes may range from one to all nodes[1]. Production systems using quorum replication (Amazon Dynamo, Apache Cassandra, Voldemort, Riak) execute reads only on a majority of nodes. Different approaches to selecting the specific majority include network proximity [42], performance history [177] or load balancing [44]. In some cases even some nodes that are part of the majority may not perform the actual read of data and rather rely on metadata (e.g. digest reads [43]). In primary-backup (PB) [65] replication implemented by several popular open-source stores (Mon-

[1]A small read-quorum typically requires a large write-quorum, overlapping in at least one replica, for ensuring strong consistency.

goDB, CouchDB, RavenDB), client read requests are served only by the primary or (relaxing consistency) by any single replica. These choices are justified by the need to optimize performance in read-dominated workloads. The system presented in this chapter is applicable to data stores serving reads from any single replica at a time.

A write operation typically requires a number (k) of acknowledgements that replicas have durably stored an update, before informing the requesting client of a successful operation. In several systems, a write request is sent to a broader set of replicas (n, where $n > k$) than the number of acknowledgements needed (k), eventually reaching the entire replica group. Updates to the first k replicas to respond are synchronous with the user update, whereas the remaining updates can proceed asynchronously in the background. Although one could argue that dissemination of updates eventually reaching all replicas (as performed by most replication technologies) could be sufficient to keep memory caches of all replicas fresh, in reality the data being written may not overlap at all with the read working set of application (e.g., in the case of writing new data while performing read-intensive analytics on past values). Thus update propagation by itself cannot maintain up-to-date memory caches across all replicas. In addition, occasional restarting of replicas, such as in the context of rejuvenation [125] and proactive recovery in adaptive replicated services [80], completely wipes out their memory caches.

Reads may shift to replicas with outdated caches. Non-read-serving replicas are not aware of reads and thus cannot keep their memory caches warm. We previously described that systems may dynamically decide which replicas form quorums that execute requests. Another reason that triggers a shift in read-serving replicas is failures. If a serving replica fails or is brought down for maintenance, a previously non-serving replica must take over. Another reason is workload migrations, where parts of an application are moved across the infrastructure for load balancing or other maintenance operations (in such cases reads are usually directed to the nearest replica). Failover actions are frequent in large data centers [60], and adaptation actions are increasingly used for performance improvement. Geo-replicated stores automatically reconfigure the set of serving replicas (e.g. leader change) to satisfy application-defined constraints as locations and their access pattern shift across different time zones [53, 138, 166, 189]. Production systems at Google [166]

trigger a reconfiguration as often as every 30 minutes when the system re-evaluates the optimal placement of a leader or its replicas based on workload characteristics, or every 2 hours in the case of Microsoft's store [53] that adapts to the shift of traffic across different time zones. Frequent reconfiguration actions may also be used as a lightweight adaptation mechanism to hide internal performance bottlenecks or in the case of colocated resource intensive background tasks on the serving replicas [96, 152]. In all these adaptation mechanisms, the newly assigned read-serving replicas (although consistent at the level of persisted data) may have missed recent reads and thus have an outdated memory cache leading to a performance impact (Section 4.3). Empirical evidence of the challenge addressed in this chapter (and inspiration for this work) is provided by our own previous research [96,153,170]. Figure 4.1 (from [152]) demonstrates that the action of changing the primary replica can hide the performance impact of a backup task, however the improved system (Figure 4.1b) still suffers from a smaller but non-negligible performance hit (area in red circle) due to cold-cache misses at the new primary (a previously non-read-serving replica). Other work on measuring the recovery time of a replicated version of the HDFS metadata server [170] (§4.4.3) showed that switching the primary to a new replica with a cold memory cache can lead to significantly higher time to recover compared to a version switching to a hot spare.

Our approach aims to keep the caches of future read-serving replicas warm by disseminating read hints to them from current read-serving replicas. This approach is similar in spirit to prefetching in storage systems, which rely on high-level knowledge of future data accesses disclosed by an application [155] or history-driven predictions [76] to warm a cache ahead of time and thus increase hit rates. Our approach differs in that instead of informed guesses, replicas learn of read operations executed on a read-serving node and execute them locally to maintain an up-to-date cache view. Traditional prefetching mechanisms are complementary to our approach and can be applied in read-serving and non-read-serving replicas. Challenges investigated in the context of prefetching, especially in balancing prefetching with caching [66], are also applicable in our work.

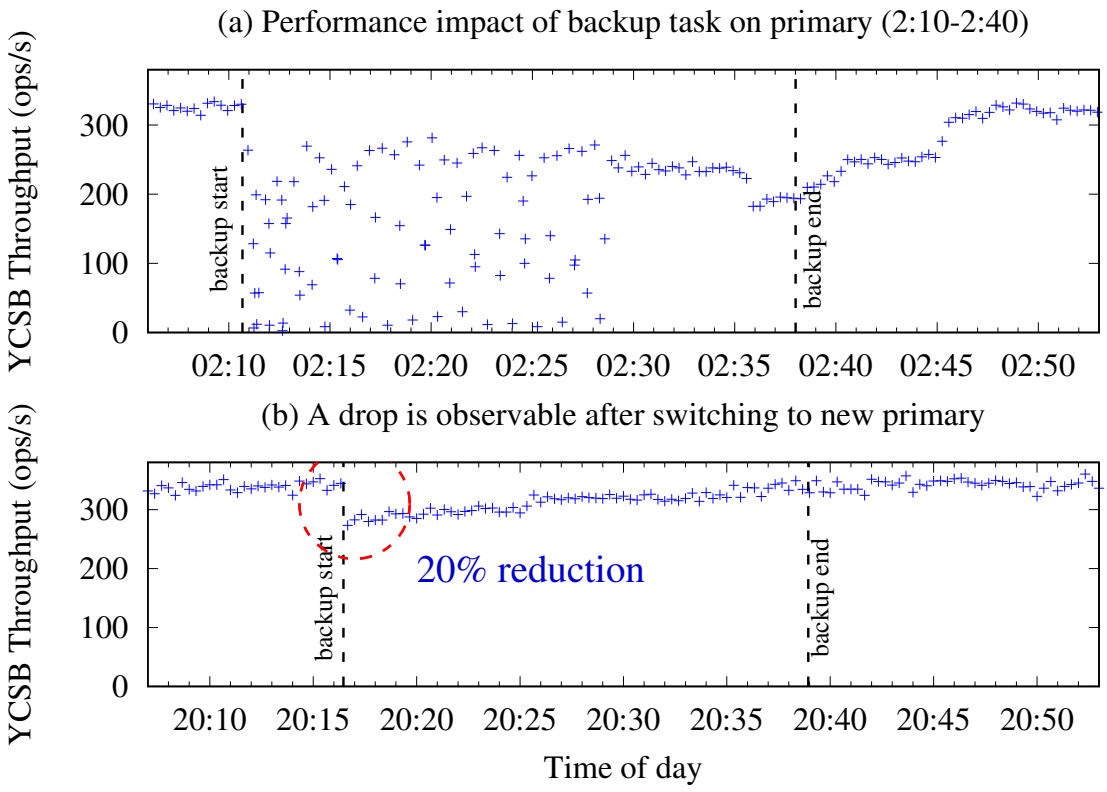


Figure 4.1: Performance impact of backup activity (a) on replica group (shard) can be hidden via reconfiguration (b), however new primary suffers from cold-cache misses [152]

4.2 Design and Implementation

We next describe the major design choices and the implementation details of our mechanism.

Design. Our goal is to ensure that non-read-serving replicas within a replica group keep track of the read working set and are always prepared to serve read requests without a performance impact due to cache misses, ensuring a smoother transition during a reconfiguration or failover. We aim to achieve this without modifying the replication protocol or system properties such as data consistency or availability. The mechanism should be transparent to users, and not have an impact on common-case performance of client read operations.

Our evaluation (Section 4.3) shows that in a typical application scenario over a replicated key-value store (MongoDB), a reconfiguration that changes the primary node in a replica group leads to higher latency for client requests, because of a low cache hit rate at the primary node, for the period of time (several minutes) it takes to load the working set into memory. This time depends on the amount of state that needs to be brought into memory (§4.3.3) and the speed at which the underlying storage device operates (§4.3.1). We aim to mitigate this problem by allowing the memory caches of non-read-serving replicas to keep track of the read working set using principles that can be easily implemented and integrated into existing replicated key-value stores.

A replication module is typically responsible for replicating requests across replicas in any distributed store, however the caching mechanism in each node is independent of the other replicas. A standard cache management policy over persisted data in each node is to apply every incoming read or write operation through the memory cache (implementing the necessary miss handling and write-through actions).

In order to disseminate read requests even to non-read-serving replicas and thus help them keep their caches warm, we propose recording the read operations executed on the serving replicas as *hints*, and send them over to non-read-serving replicas in the background. To achieve this, we introduce a *read-hints module* (RHM) and integrate it into all replicas. Figure 4.2 presents the RHM integrated into the primary and sencondary replicas. Read-serving nodes use RHM to passively monitor the executed read requests and maintain a *read-hints buffer* (RHB), which keeps a minimal description of each read operation applied on the serving replica. RHM periodically disseminates the contents of the buffer (in batches) to the non-read-serving replicas which store the received hint into their local RHB. Based on the hints a replica is able to replay read-operations (note that we are shipping read operations, not the data), leveraging the data store's existing read path and cache management mechanism.

Our mechanism does not require any modifications to the pre-existing replication and dissemination of writes, it uses separate data structures and dissemination channel for the read hints as depicted in Figure 4.2 and in no way affects the data consistency or availability properties of the system (§4.2.3). As a read operation can always be satisfied by the

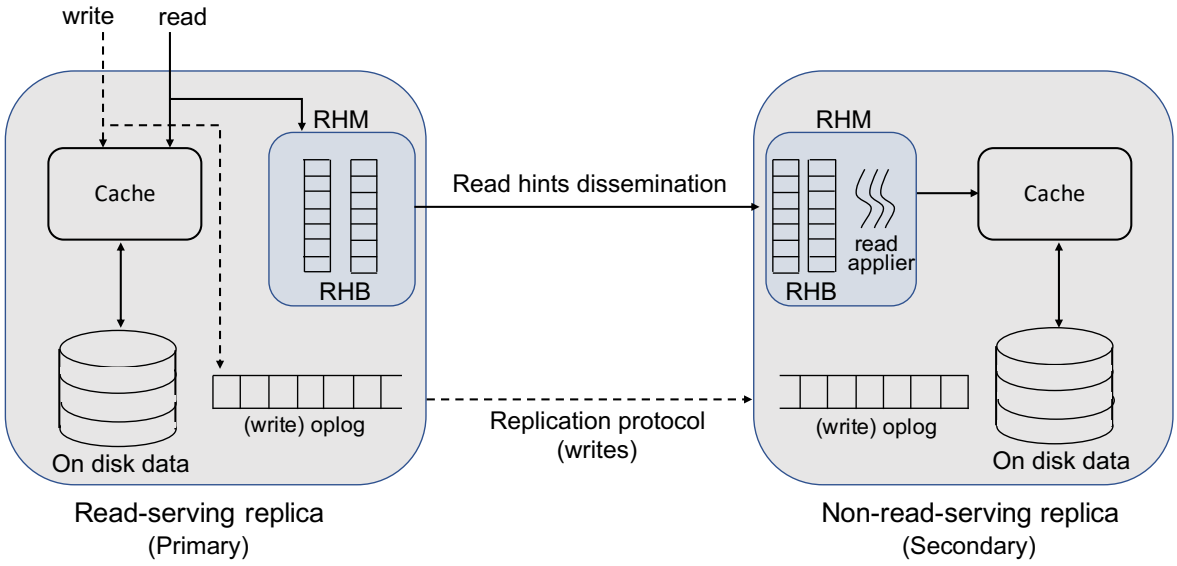


Figure 4.2: The Read-Hints Module (RHM) is integrated as a plugin into the replica-maintainance path; this particular figure is based on MongoDB internals. The module passively monitors read requests and maintains a Read-Hint Buffer (RHB) that is periodically disseminated across replicas

underlying disk, our proposal is a *best-effort* mechanism that allows replicas to keep their caches warm when they become aware of the reads executed on the serving nodes. The design supports dissemination of read operations from more than one read-serving replicas and allows tuning of the degree of dissemination (how many replicas to disseminate to) and intensity of communication (degree of batching and frequency). In systems where reads are served by a different replica at a time for load balancing, RHM can be configured so that each read-serving replica disseminates read-hints to all replicas. However, we expect the biggest benefit from RHM will be realized in strongly-consistent systems that serve reads from a single primary replica.

A naive implementation that directly issues read requests to all replica nodes and waits for the first response is expected to maintain warm caches across all replicas. However, the consistency model of such approaches is weaker compared to the strong semantics achievable by reading and writing from/to the primary node. As such, naive approaches are not an option for applications requiring strong semantics from the underlying data

store, whereas RHM covers such cases as well. RHM is based on an asynchronous best-effort background dissemination of read-hints and does not require any modification to the read/write protocol.

Section 4.2.1 describes in detail the implementation, tuning parameters, and different tradeoffs involved extending MongoDB v4.0.6. Section 4.2.2 supports our choice to design our mechanism as a separate module of the data store and discusses read-hints buffer properties. Section 4.2.3 explains that the design does not affect the replication and consistency model of the data store and that data consistency is preserved between in-cache and on-disk data within a replica node that updates its cache based on the received read hints buffer.

4.2.1 Capturing and dissemination of read hints

Our implementation of the read-hints module extends the MongoDB v4.0.6 codebase. MongoDB implements a passive replication system following the primary-backup replication model. The primary node serves client requests. In case of writes it applies the operation to its local state and updates its cache. Each update is also logged to its internal *oplog* data structure (Figure 4.2). The updates are propagated to all replicas through the replication of the oplog. However the RHM is by design a separate module. The rationale for this decision, as opposed to integrating reads into the existing *oplog*, is discussed in Section 4.2.2.

The RHM monitors reads served on a primary extending the code path that handles the read request, cloning the read operation and channeling it to the RHM code. It keeps into RHB a minimal description of each read operation to allow a replica to locate the data, avoiding unnecessary information such as request type, session ID and hash signatures used for sanity checking on the primary node only. The RHB is periodically synced across replicas, with internode communication within a replica group specifically built for this purpose and implemented over TCP/IP sockets. The replication is performed by default in batches over 90ms intervals or when the RHB reaches its maximum capacity (by default 10,000 entries). We have empirically determined that these settings allow replicas to be reasonably up-to-date with minimal overhead (§4.3.7), but both parameters are

configurable. We experiment with different settings in Section 4.3.

On a read-serving node, a background thread is responsible for periodically sync'ing the RHB to the replicas. To avoid contention between threads that update and disseminate RHB across replicas we apply double-buffering techniques, i.e. when the buffer is marked as ready to be shipped over the network, it is marked as read-only and swapped with a new empty buffer to log the read-hints. We also pre-allocate the memory buffers that support RHB to avoid allocation overheads. After the RHB synchronization is complete, the read-only buffer is marked as clear and ready to be reused.

Batching multiple reads allows for summarization, an optimization where multiple occurrences of the same request in the RHB may be consolidated to the last read of each key. Thus several requests for the same data are logged just once in the buffer, enabling a compact view of the requested data and reducing the total RHB size to be shipped across replicas. Other techniques [192] that further reduce the amount of transferred data over the network could also be applied.

On the receiving replica, a receiver thread, part of the RHM, is responsible for handling incoming connections from the serving node RHM and storing hints into its local buffer. Threads off of a read-applier thread pool are in charge of taking the read operations off the local RHB and applying them to the local copy of the database. During periods of resource strain, a secondary replica can apply a subset of read hints received or even turn RHM off while overload conditions last (§4.3.8). Threads from the read-applier pool shepherd requests through the same code path followed by a normal user read request (except this code path was previously only executed by the primary replica). In this way, the replica cache tracks the read working set along with the primary node's cache.

RHM allows the configuration on the number of replicas that will sync the RHB. It supports three options: *all, one, region-one*. With option *all* (*one*), all (one) replica(s) receive RHB updates from the primary node. Option *one* is useful when CPU and network bandwidth are scarce; on the downside, only one replica will be well prepared to serve reads. *Region-one* is used in geo-replicated systems and allows syncing the read log with one replica in each region. Furthermore in resource-constrained nodes where secondary replicas are co-located with primaries, the RHM can selectively apply or even turn off read

hints, similar to adaptation strategies followed in prefetching systems [66].

When reconfiguration actions are known in advance, a potential alternative is to delay the dissemination of read hints to that time. However, such a delay is not expected to be beneficial. Storing the history of read hints at the read-serving node for a long time would require significant amounts of memory. In addition, disseminating read-hints in a burst just before a reconfiguration action could pose a significant performance hit and also delay the reconfiguration until the replica is prepared. Thus we believe that even in the case of frequent reconfiguration actions (such as in systems described in Section 4.1), there are practical benefits to the continuous dissemination of read-hints.

4.2.2 Read-hints buffer properties

Our mechanism is by design a self-contained module (Figure 4.2), ensuring that our implementation is generally applicable (not too tied to the design principles of a specific stores replication module) and can be easily integrated into most distributed data stores. Using existing data structures or mechanisms of the existing replication subsystem would require tight coupling with a specific store. In addition, our read hints buffers have different durability properties. Since the cache is essentially *soft state* that need not be persisted as a separate entity, we can treat the dissemination of the read as a hint mechanism to the replicas; thus in case of node failure, a lost RHB has no impact on data availability or recovery mechanism. This also allows to delay the replication of the hint buffers and transfer read ops in batches to amortize the transmission cost. Batching allows us to benefit from summarization and to further reduce the amount of transferred data (§4.3.10).

The format of an RHB entry is simple: it carries only minimal description of the requested data (e.g. the key and some filters) derived from a read request –there is no need to include information about client, session or timestamps as described in Section 4.2.3– further contributing to a reduction of the amount of information sent over the wire.

4.2.3 Consistency

Consistency across nodes. Our mechanism does not affect the store's consistency model leaving the replication mechanism intact. Standard replication mechanisms dictate how committed state updates are eventually applied to all replicas. Updating replica caches by our read-operation dissemination mechanism ensures that a cache is refreshed with the most recent state that has already been applied on the same replica (stored on its disk). The shipping of read-operations (rather than shipping the data) and their execution, reflects the latest state known to replicas in their local cache. As such, the cache remains consistent with the on-disk state, which in turn follows whatever consistency guarantees are implemented in the specific distributed store.

Consistency between in-cache and on-disk state. As reads are shipped independently of writes across replicas, there is a probability that reads may be reordered relative to writes (Figure 4.3). This does not cause a problem as a read operation applied in the local cache of another replica will always reflect in memory the latest state written to disk at the time. In this way, cache contents always remain valid. The order of writes will remain as decided by the primary, and each write will consistently update cache and disk.

We note that correctness relies on correct implementation of atomic execution and serialization of read and write (updating cache and disk) operations in each replica's storage backend.

Cached objects view across nodes. Even with our mechanism, caches in different nodes may have different contents due to the delay in disseminating operations as well as due to actions of the cache replacement policy. Typically, most systems feature a cache eviction policy that favors recently or frequently accessed objects (e.g., LRU, LFU). We expect that with our mechanism, a cache will contain the objects of the last applied RHB and the most recently accessed objects for any node (with the primary or read-serving nodes being somewhat ahead of others). It is not a goal of our work to keep caches fully in sync, and we think that there is little benefit in trying to achieve such a stringent objective. Our approach is a best effort mechanism aiming to help non-read-serving replicas track the read working set and thus be well prepared to serve future reads. In Section 4.3 we show

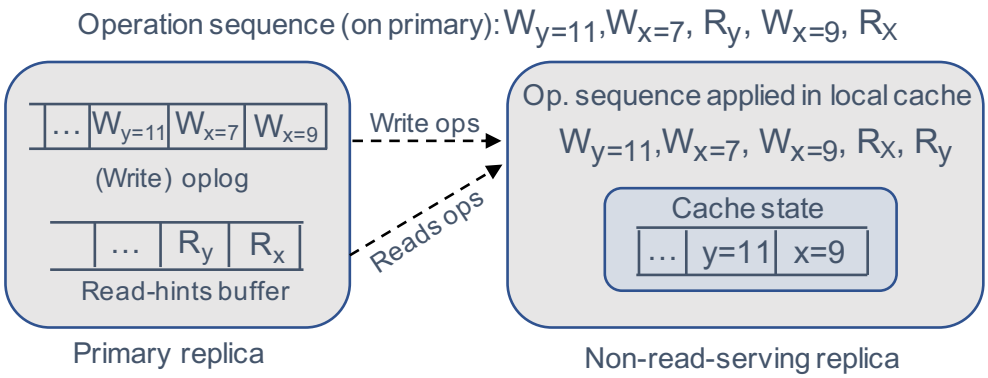


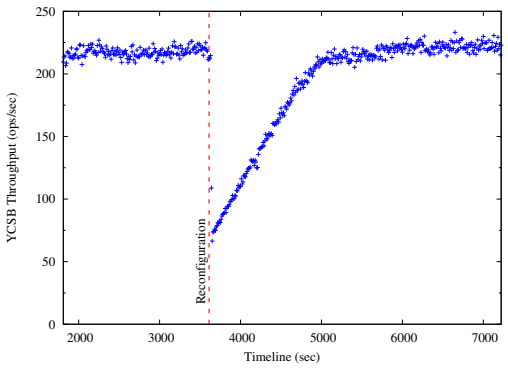
Figure 4.3: Reads may be re-ordered relative to writes, however caches always contain the latest state written to disk

that our mechanism is able to achieve our goal with no noticeable overhead.

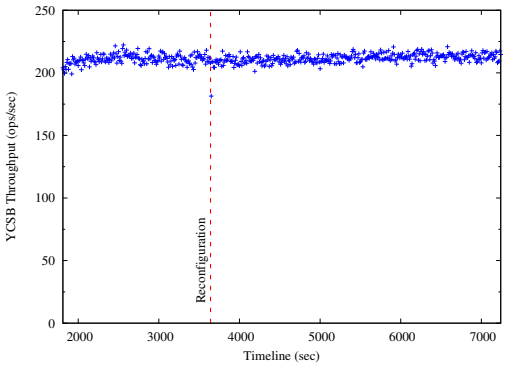
4.3 Evaluation

In this section we evaluate the benefits of allowing non-read-serving replicas track the read working set using our extended prototype of MongoDB v4.0.6. The MongoDB binaries used have been compiled with debug symbols. Our results show that the system exhibits stable performance after shifting the node that serves reads from primary to the nearest secondary replica after the workload migration. The overhead of our mechanisms does not have a measurable impact on overall system performance (compared to the baseline implementation). Unless otherwise stated, all experiments use a non-sharded MongoDB installation with one replica group consisting of one primary and two replica nodes. Our main experimental testbed consists of four servers, each equipped with a Intel Xeon Bronze 3106 8-core 1.70GHz CPU, 16GB DDR4 2666MHz DIMMs, 256GB Intel D3-S4610 SSD and 2TB Ultrastar 7K2 HDD, running Ubuntu Linux 16.04.6 LTS, interconnected via a 10Gb/s Dell N4032 switch. Whenever a different testbed is used (such as AWS EC2 in §4.3.2), this is clearly stated in the text.

Our evaluation includes three workloads: the Yahoo Cloud Serving Benchmark (YCSB) [81] v0.11, TPC-C [38], a popular OLTP benchmark adapted for NoSQL systems [118]; and an

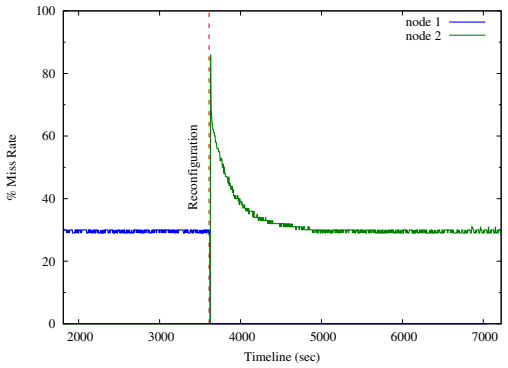


(a) Read-hints module disabled (unmodified MongoDB)

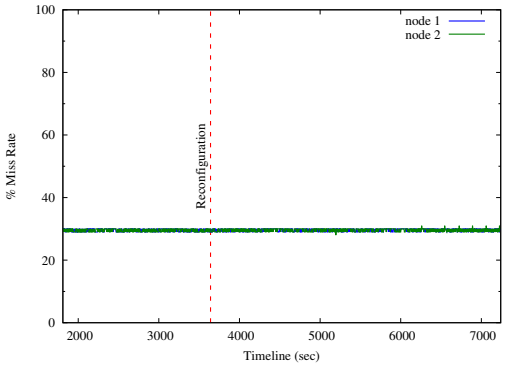


(b) Read-hints module enabled

Figure 4.4: Throughput under read-only workload, HDD used as back-end store



(a) Read-hints module disabled (unmodified MongoDB)



(b) Read-hints module enabled

Figure 4.5: Cache miss rate under read-only workload, HDD used as back-end store

analytical processing workload modeled after the popular TPC-H [39] benchmark. In all cases the workload driver executes on a dedicated server. In experiments where the workload switches the read-serving node to the nearest replica (to achieve the shortest possible latency), we sidestep the fact that our testbed has a flat network topology (nearly identical latency across all replicas) and emulate the selection of the nearest replica via MongoDB's server tagging feature [25]. The dataset is loaded fresh onto MongoDB nodes before each experiment and caches are dropped before each run.

4.3.1 Performance impact during reconfiguration

In this section we evaluate our method under the YCSB workload generator. The benchmark is configured to produce two different mixes of reads vs. updates/writes: a 100% read workload to stress our mechanism as the primary node has to keep up with a busy read-hints module, and 80% (reads)-20% (updates) to demonstrate performance under a mixed workload including both reads and writes.

Unless otherwise stated, the requested keys are randomly selected from a uniform distribution. The dataset consists of 50 million unique records resulting to 60GB of data on disk (including indexes) per node. The number of YCSB client threads (number of parallel connections between database client and servers) is set to 8, empirically determined to stress the cluster while keeping average response time under 20ms (considered a reasonable threshold). MongoDB is configured with 10GB of cache size. Read and write concern use the *majority* option by default.

Figure 4.4 depicts YCSB throughput (ops/sec) under the read-only workload, with data stored on HDDs on each node. As in all subsequent figures, time (x-axis) starts 30 minutes into the experiment, when the system is deemed to have reached a steady state. The y-axis depicts throughput in YCSB ops/sec. Figure 4.5 presents the MongoDB cache miss rate on node 1 (the primary node that serves client requests at the beginning of the run), and node 2 (the nearest replica serving reads after the workload migration). Figures 4.4a and 4.5a correspond to unmodified MongoDB, while Figures 4.4b and 4.5b correspond to our prototype. The vertical dashed line represents the transition of read serving node from node 1 to node 2.

Figure 4.4a exhibits a clear performance hit for unmodified MongoDB (read-hints module is disabled), right after the reconfiguration at 3600 seconds. We observe system throughput (217 ops/sec before the reconfiguration) dropping to 65 ops/sec right after reconfiguration (a 70% reduction), taking 18 minutes to reach its pre-reconfiguration serving rate again. The performance hit is explained by the new serving node (node 2) high cache miss rate (up to 86%) at the early stages of its serving operation phase (Figure 4.5a), significantly higher than the 31% miss rate that either node experiences while at steady state. Note that

the reported cache miss rate takes into account data and index accesses, as MongoDB loads indexes into its internal cache to speed up read requests. In Figure 4.5a, node 2 reports 0% cache activity before it starts serving client requests (as it does not perform any read operations); similarly node 1 does not serve reads after the workload migration starts directing its requests to the nearest replica. We note that had node 2 been operating as a read-serving node in the recent past, the performance impact may be lower than observed in this experiment.

Figure 4.4b depicts throughput with our RHM enabled. The reported throughput is about 211 ops/sec during the whole run of the experiment, while the MongoDB cache miss rate remains stable at 31% at all nodes during the entire run (Figure 4.5b). We observe that our mechanisms are effective in keeping replica caches up to date, saving a 55% spike on the cache miss rate experienced by unmodified MongoDB when the nearest replica starts serving client requests. Unmodified MongoDB cannot meet its pre-reconfiguration throughput for a long period of time, even with a cache size of 10 GB.

For further insight into internal resource use, we report the total amount of physical memory used in each node (Figure 4.6). In line with observations in Figure 4.5b, it takes almost 20 minutes to bring data in memory right after reconfiguration at 3600 seconds with unmodified MongoDB (Figure 4.6a, node 2). Our prototype (Figure 4.6b) is able to maintain data in memory (application cache) in node 2, resulting in a smoother transition to the new configuration. Note that the memory reported as used exceeds 10GB (the MongoDB cache size) as it includes the memory used by the MongoDB process and the OS as well.

Next we study the effect of a faster storage device (SSD) on the performance impact during reconfiguration. In general, using SSDs we expect to be able to serve client requests and to recover (restore a replica's working set) at a higher rate. Figure 4.7a depicts throughput with unmodified MongoDB using SSD as a back-end store. The system initially serves requests at a rate of 2260 ops/sec. At 1250 seconds we trigger the workload migration and the serving replica moves from node 1 to node 2, which causes a clear performance hit. The throughput drops at 1510 ops/sec (a 33.1% reduction), taking over 2 minutes to reach its stable state again of 2260 ops/sec, caused by cold-cache misses at the new serving node.

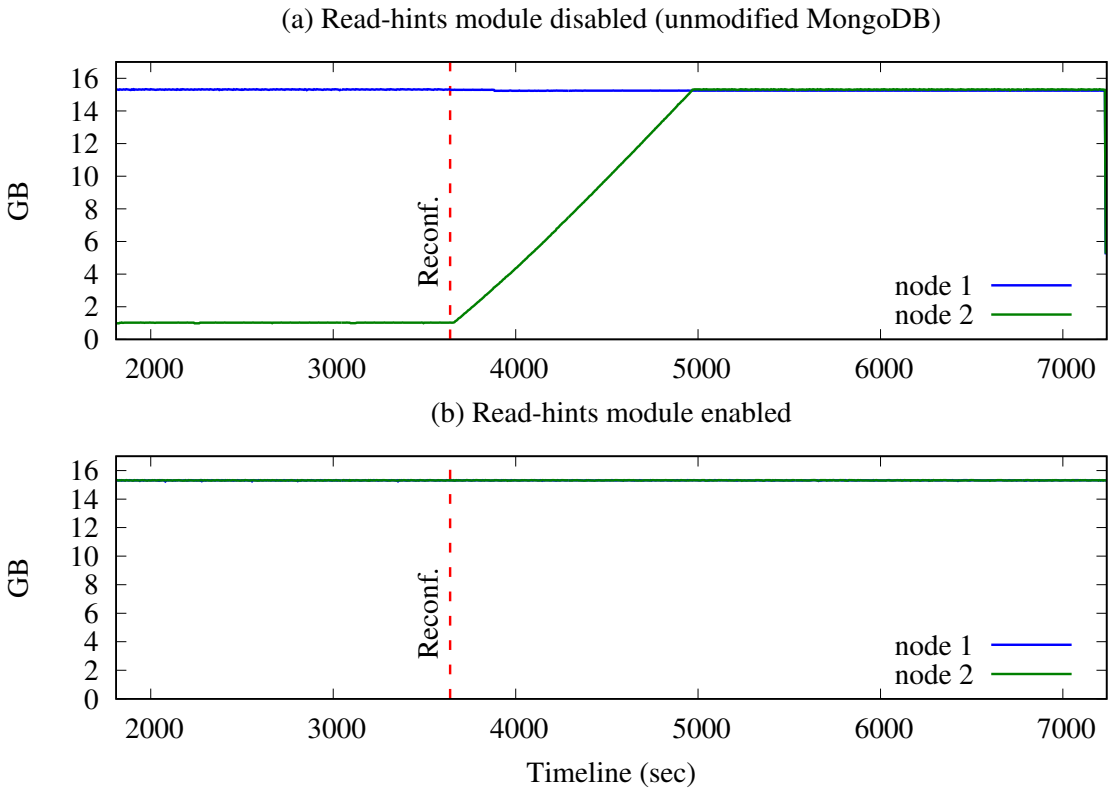


Figure 4.6: Monitoring memory use

Figure 4.8a depicts MongoDB cache experiencing a 87% miss rate right after reconfiguration, before reaching again its stable rate at 31% until the end of the run. Our prototype is able to maintain throughput stable during the whole run (Figure 4.7b) as the MongoDB cache-miss rate is not affected when moving the serving replica from node 1 to node 2 (Figure 4.8b).

We note that with the read-hints module disabled, MongoDB cache-miss rate at the early stages of reconfiguration does not depend on the back-end store (HDD or SSD). This is expected as the dataset and cache capacity are the same in both cases. In the case of SSD, the system is able to recover faster as it can serve client requests at a higher rate. However, recovery time would increase as cache sizes grow, as seen in §4.3.3. Overall we observe that even with faster storage devices, a primary-backup replication management system that executes reads only at the primary, exhibits a steep performance degradation for long

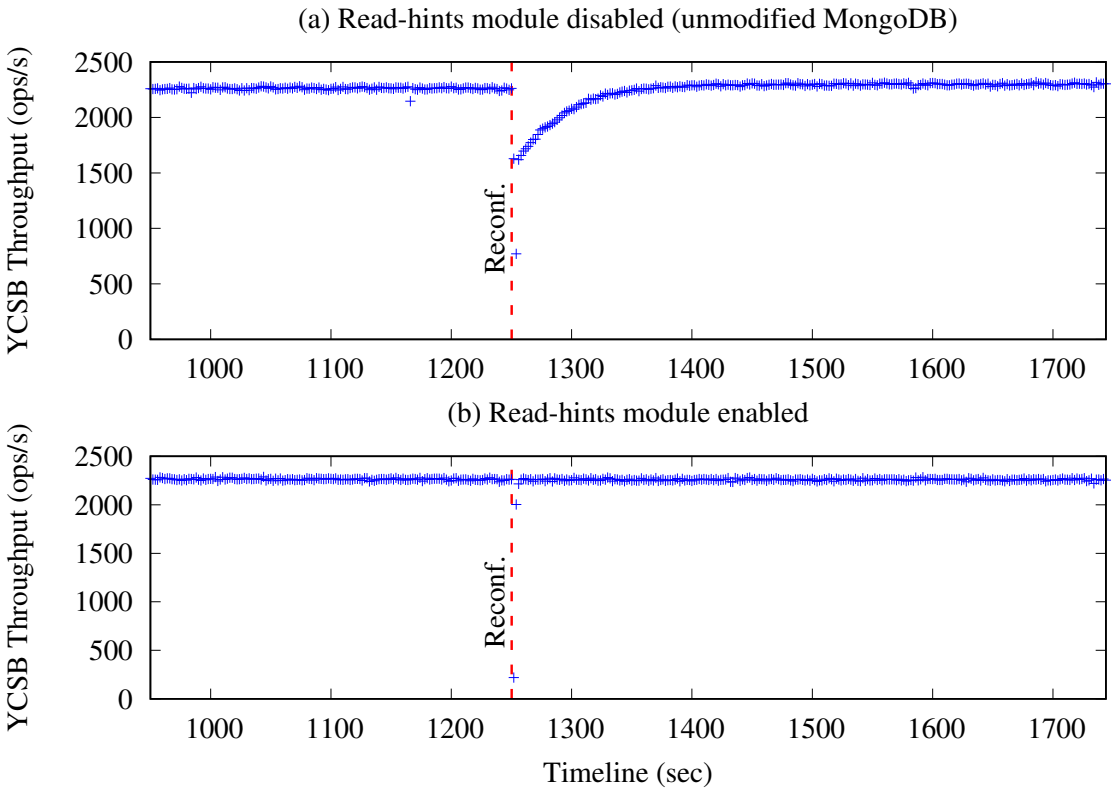


Figure 4.7: Throughput under read-only workload, SSD used as back-end store

periods of time (several minutes) right after reconfiguration, due to cold-cache misses at the new serving node. Our mechanisms are able to mask this impact with no measurable performance overhead in steady-state performance.

4.3.2 Multi-shard deployments on AWS EC2

Next we evaluate the RHM mechanism over a sharded MongoDB deployment using multiple replica groups. The evaluation was carried out on Amazon EC2 cloud platform, to validate portability and reproducibility of our results. Our database is split in three shards. Each shard comprises a replica group. Each server node hosts two MongoDB instances, a primary and a secondary replica, belonging to different shards. A multi-shard deployment requires a MongoDB metadata config server and a query router process (mongos), inter-

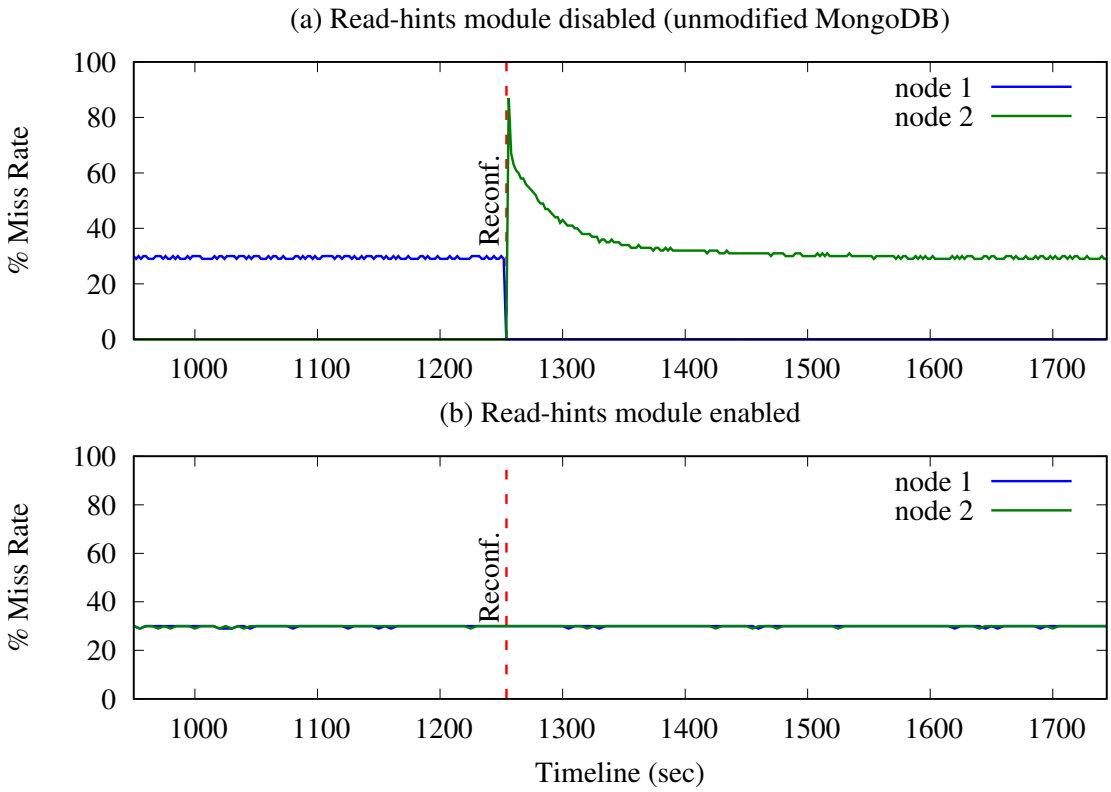


Figure 4.8: Cache miss rate under read-only workload, SSD used a back-end store

facing between client applications and the sharded cluster[2]. The 3 EC2 VMs allocated for the database nodes are of type r5a.xlarge featuring 4 vCPUs and 32GB of memory. Each MongoDB process is configured with 14GB of cache size. The metadata server is hosted on a c5.xlarge (4 high performance vCPUs) instance. We use a dedicated c5.xlarge instance for the YCSB workload generator.

Figure 4.9 depicts the aggregate throughput of unmodified MongoDB vs. with RHM enabled. In the case of unmodified MongoDB (top graph) we observe a clear performance hit during reconfiguration, with system throughput dropping from 1090 ops/sec before reconfiguration to below 500 ops/sec right after reconfiguration, requiring almost 2 minutes to fully recover to the pre-reconfiguration throughput. MongoDB with RHM enabled exhibits a smooth transition to the new configuration (Figure 4.9b). The cache-miss ratio

[2]https://docs.mongodb.com/manual/sharding (retrieved july 2021)

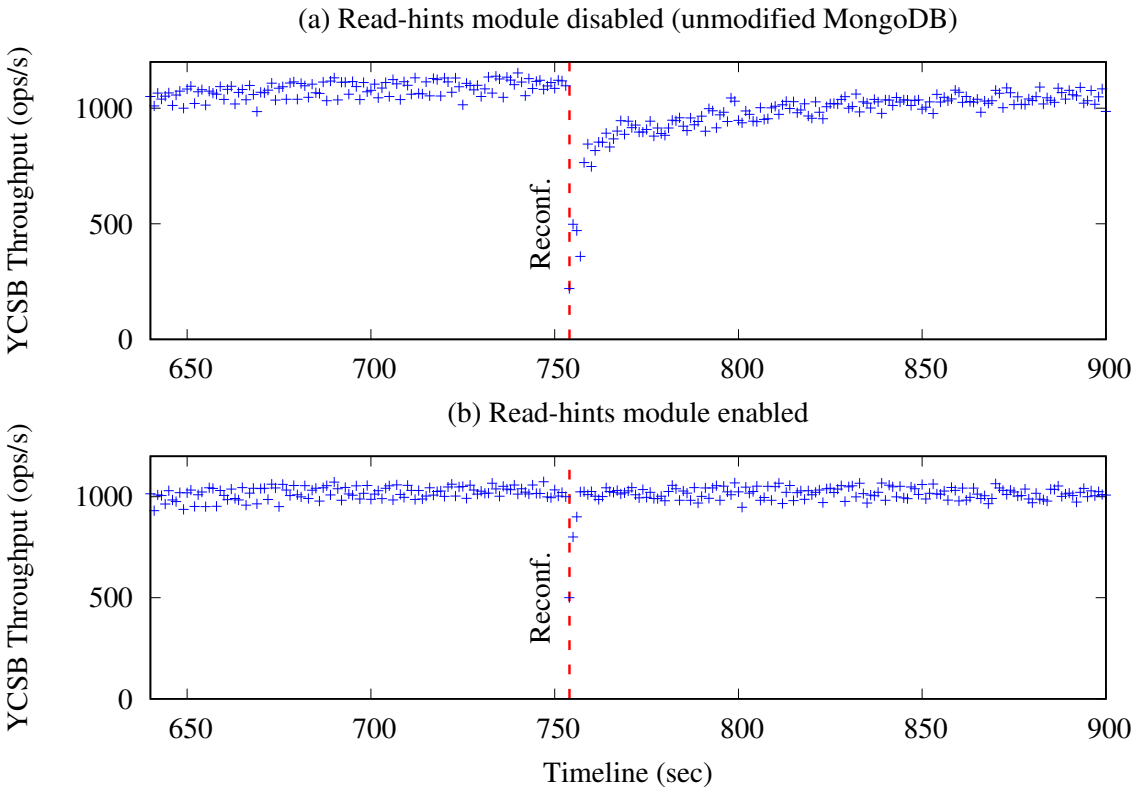


Figure 4.9: Aggregate throughput of a multi-sharded replicated database on AWS EC2 under read-only workload

for unmodified MongoDB is up to 75% for the new primary node right after the reconfiguration, exhibiting a similar trend to the cache-miss ratio of experiments in §4.3.1.

The aggregate steady-state throughput in this experiment is lower than reported in the experiments of §4.3.1. This is due to different specs of the two platforms and the additional entities (metadata configuration server and query router) required for a multi-shard MongoDB deployment.However, both cases exhibit a similar trend: unmodified MongoDB suffers a significant performance hit due to cold-cache misses after a reconfiguration, whereas MongoDB with RHM enabled results in a much smoother transition.

4.3.3 Effect of cache size on time to restore performance

In previous experiments we noted that the time to restore performance of unmodified MongoDB after the read-serving node changes, improves with a faster back-end store. In this section we investigate experimentally the impact of the size of the cache, an important issue in light of larger memory capacities typical of high-end enterprise servers. To evaluate the relationship between the time to restore performance after a read-serving replica change vs. cache size, we perform experiments using unmodified MongoDB with an SSD back-end and different cache sizes, measuring the time to reach its pre-reconfiguration throughput after a primary change.

To ensure that we can control the memory available for caching, we had to regulate both MongoDB's own internal cache implemented within its storage engine (WiredTiger) as well as the filesystem cache. MongoDB' internal WiredTiger cache loads collection data and indexes and is of configurable size; however, the filesystem also indirectly caches MongoDB data and automatically uses free memory left unused by the WiredTiger cache or other processes. MongoDB thus benefits from both caches to reduce disk I/O. To effect a system-wide limit on cache size, we resorted to the Linux control groups (cgroups) feature that can limit the total memory available for all caches (MongoDB internal and filesystem caches).

Figure 4.10 shows that there is a direct relationship between recovery time (time to refill the cache) and cache size. Increasing cache capacity from 1GB to 10GB leads to longer time to refill the cache, taking over 2 minutes to reach steady state performance level with 10 GB of cache as it has to bring more data into memory to fill the cache and reach steady state. Production deployments with even larger memory capacities (orders of magnitude larger memories are very common in enterprise environments) are expected to result in much longer periods of low system performance. Large caches are expected to be characterized by low response times (during steady-state performance) and long time to refill after a reconfiguration, yielding prolonged periods of large (as a ratio of recovery vs. steady-state performance) SLO violations, making a strong case for the mechanisms proposed.

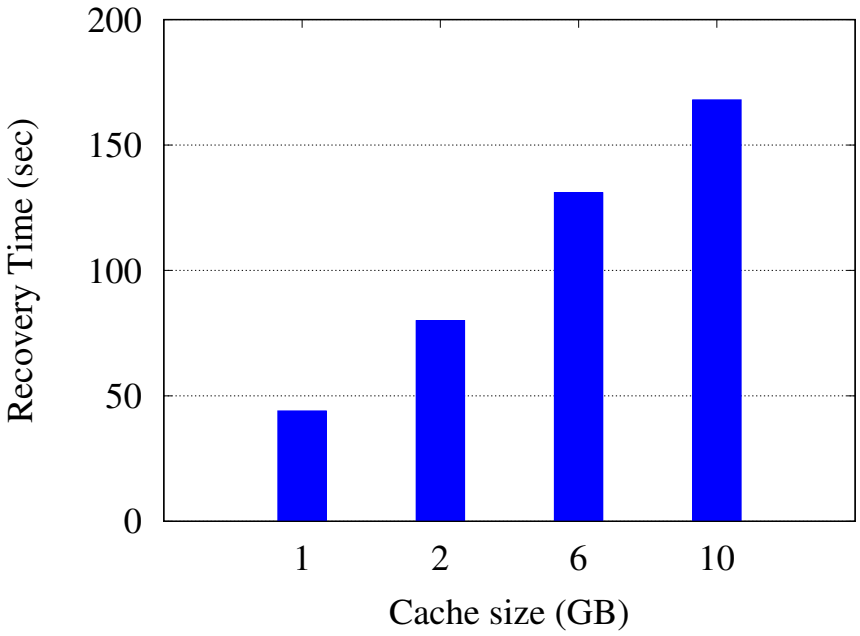


Figure 4.10: Time to restore performance level vs. cache size

4.3.4 Effect of cache access pattern

To determine what (if any) is the impact of the cache access pattern, we run experiments with the read-only workload and a Zipf-distributed (rather than uniform) access pattern. We thus configure YCSB to select keys using the Zipf distribution, which exhibits a stronger temporal locality and is expected to increase the efficiency of cache and result in higher overall throughput.

In Figure 4.11 we observe that throughput is indeed increased to 2689 ops/sec. MongoDB (both the unmodified version and our prototype) exhibits a stable miss rate at 18% over the entire run. Following a reconfiguration, the throughput of unmodified MongoDB drops by 39% due to an increase of the cache miss rate. The cache behaviour follows a similar trend to the experiments under uniform distribution (Figure 4.8). After the reconfiguration, the cache miss rate increases to 80% (Figure 4.12). As the new serving node is warming up its cache, it takes almost up to 2 minutes for the system to reach the steady-state throughput following the reconfiguration. Our prototype exhibits a smooth transition to the new serving replica, just as in previous cases. Thus, even though we have stronger

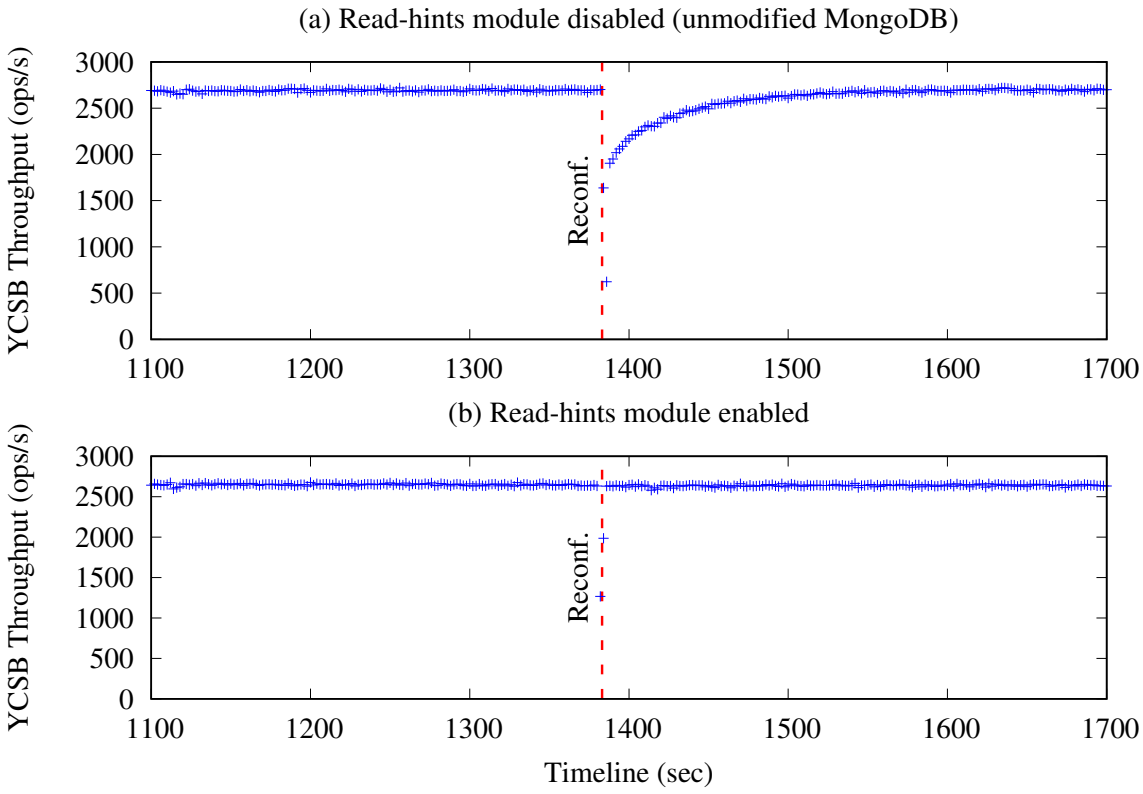


Figure 4.11: Throughput under read-only workload, Zipf distribution, SSD back-end store

locality and more efficient use of cache in this case, we observe similar behavior to the experiment with uniform distribution.

4.3.5 Read-write workload

In the 100%-read workload evaluated so far, backup replicas in unmodified MongoDB did not see any of the read operations, explaining the high miss rates experienced after reconfiguration. As soon as writes are introduced into the mix, however, backup replicas have a way to learn of updates (as writes are propagated through the stores replication mechanism). Thus, *if the working set of future reads has strong spatial and temporal overlap with the working set of past writes*, it is expected that the performance impact of a reconfiguration action will be reduced. Indeed, our experiments showed that when the percentage

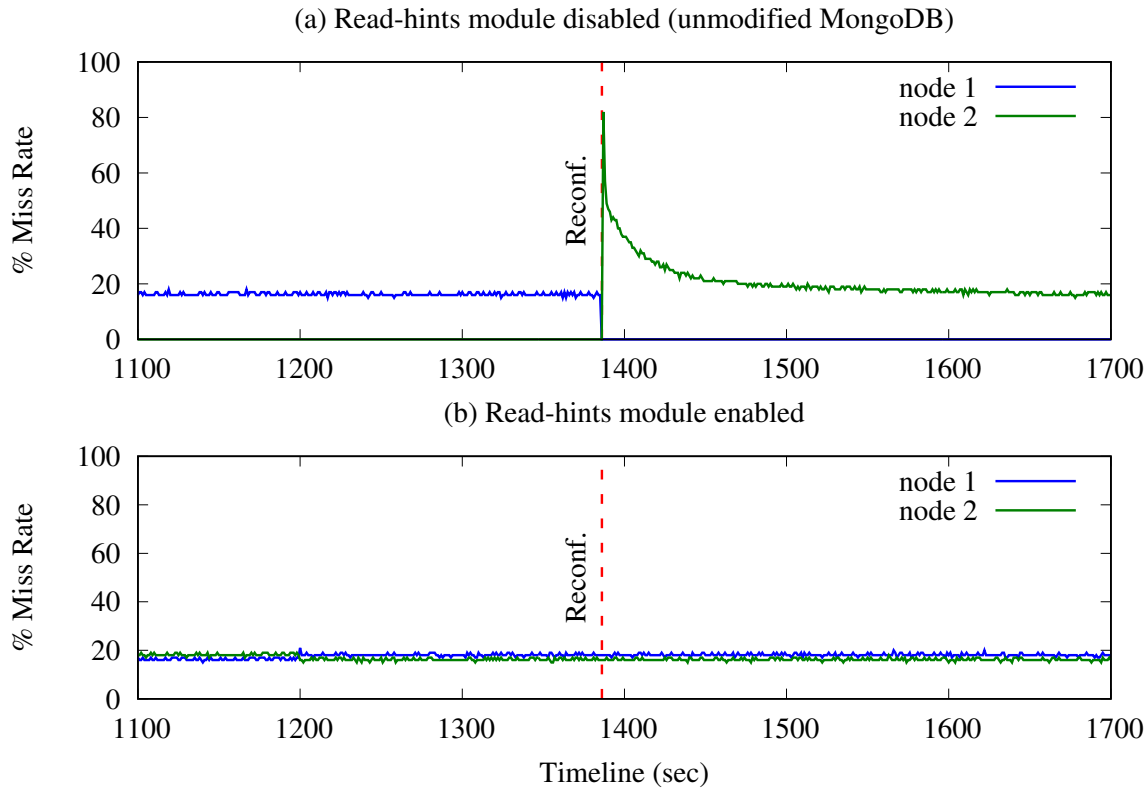


Figure 4.12: Cache miss rate under read-only workload, Zipf distribution, SSD back-end store

of writes issued by YCSB increase, the performance impact experienced by unmodified MongoDB diminishes, due to the fact that reads and updates access the same set of keys. However, in several applications (e.g., in typical big-data analytics scenarios where updates are inserting new records, whereas reads are accessing past -historical- records) this is not expected to be the case. To characterize such a scenario, we set up an experiment with an 80%-20% read-write workload mix. However, in this case the read and writes are performed on a different set of keys, using a uniform distribution for both operation types.

Figure 4.13a depicts results with unmodified MongoDB, with read throughput exhibiting a clear performance hit, up to 58% during reconfiguration. This is due to the high cache miss rate (89%) on the new serving replica right after the workload migration (Figure 4.14a). Write throughput is not affected by the reconfiguration as caching does not

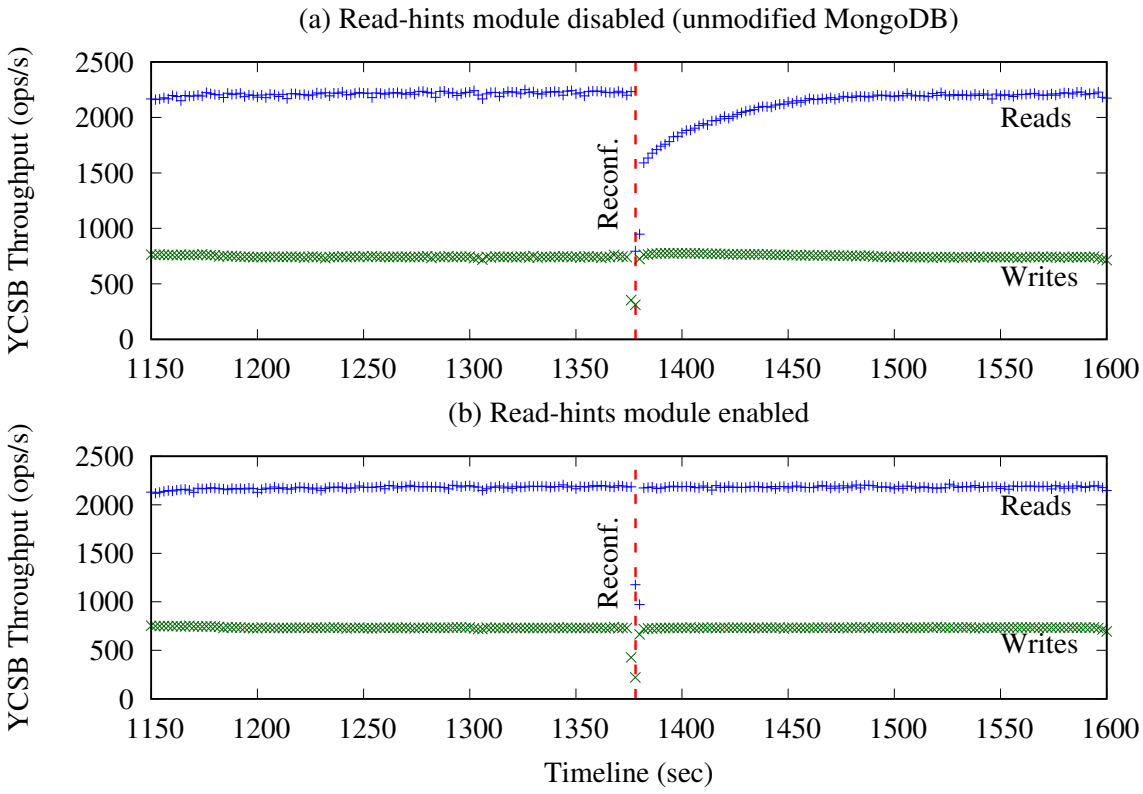


Figure 4.13: Throughput under read-write workload, uniform distribution, SSD back-end store

directly affect write performance.

In Section 4.3.11 we extend our evaluation through experimentation with TPC-C and TPC-H workloads.

4.3.6 Re-electing a past primary

In previous sections, the replica node that takes over as a primary has an empty cache as would be the case if that node recently joined the group. In a system that has been online for some time and, especially for a node that has served as a replica-group primary in the past, it may be the case that its cache has some or all of the key-values in the current working set of the client application, thus resulting in few or no cache misses after the reconfiguration. In Figure 4.15a (YCSB throughput) we exhibit the outcome of a second

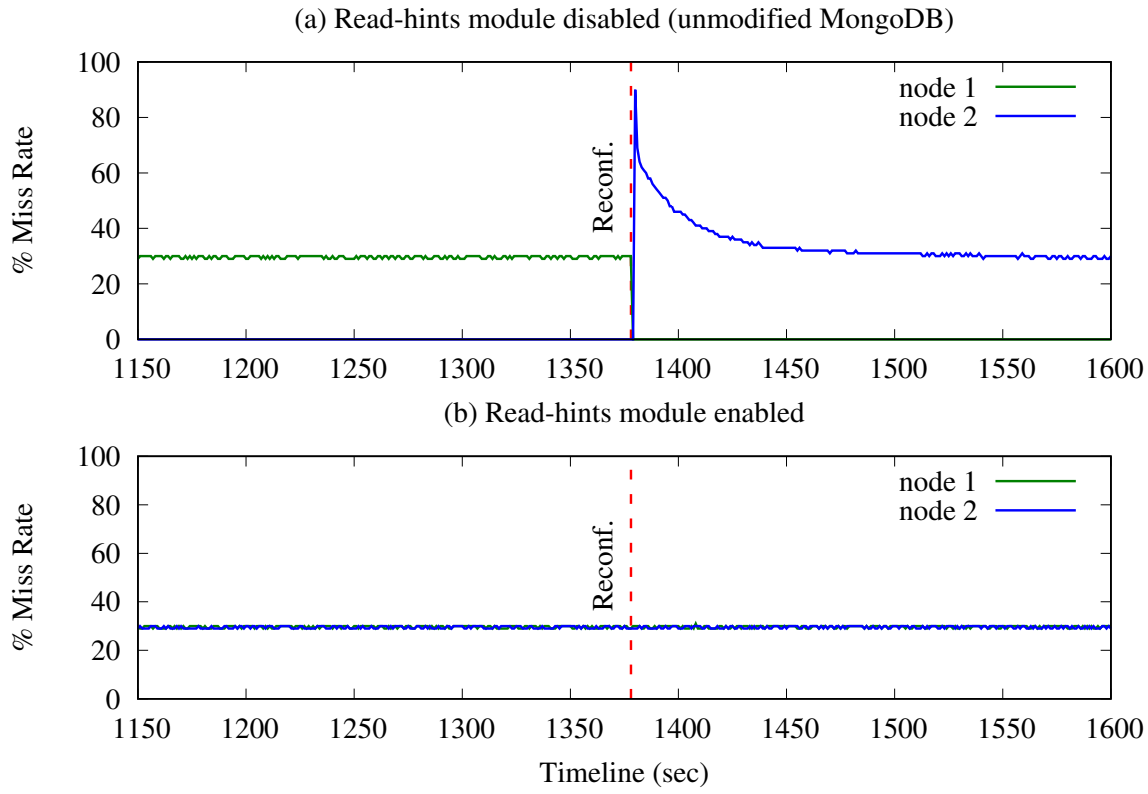


Figure 4.14: Cache miss rate under read-write workload, uniform distribution, SSD back-end store

reconfiguration re-electing as leader node 1 at 610 sec, after having served as leader in 0-300 sec, with RHM disabled. The throughput remains unaffected during the second reconfiguration as YCSB clients access the same working set during the run. In Figure 4.15b we observe that node 1's cache miss rate does not increase when node 1 becomes primary again. Several practical factors however can still render a past leader's cache cold: One such case is when an application's working set changes over time, a fact that has been supported by analyses of real environments [57]. Other factors that may occasionally wipe out the memory contents of replica nodes are (1) replica migrations that often take place in the background for load balancing or management needs, or (2) power losses and reboots.

To highlight the impact of a time-varying working set when RHM is disabled, we set

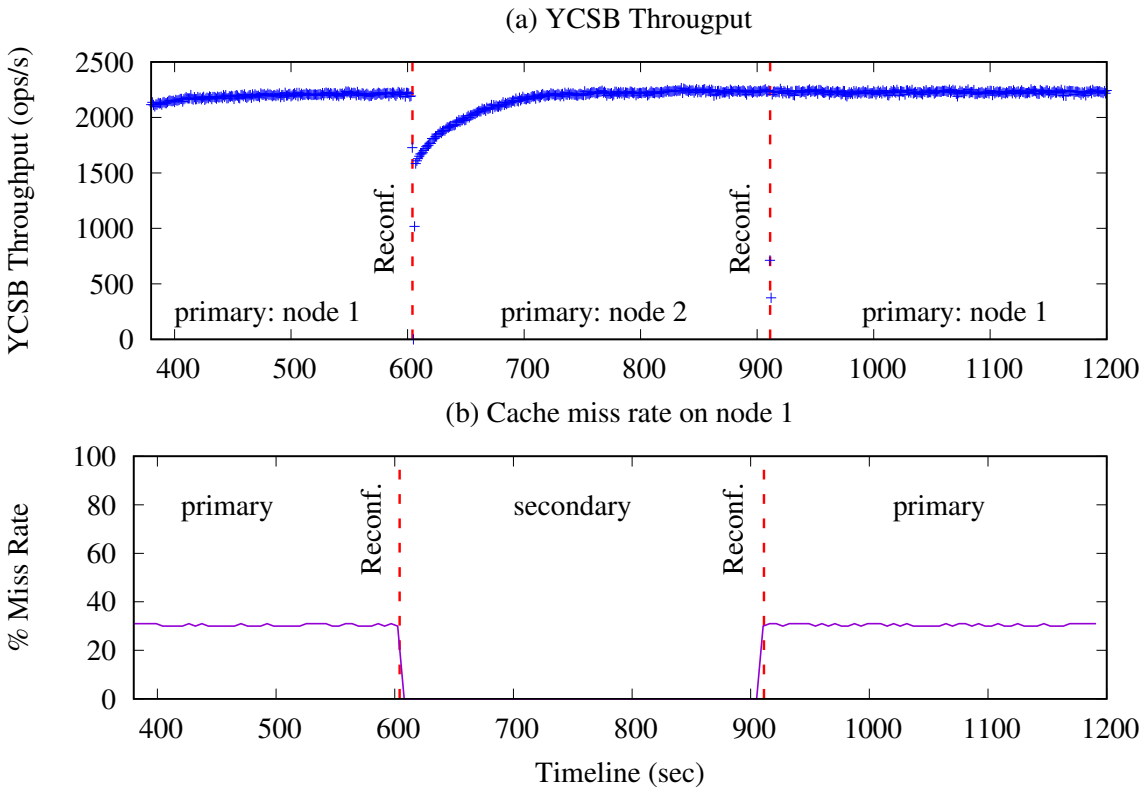


Figure 4.15: Node 1 serves as primary again at 600 sec after a second reconfiguration (RHM disabled, same working set)

up an experiment[3] featuring two synthetic workloads (two YCSB instances) with different working sets. Again we perform two reconfigurations, with node 1 re-elected primary for a second time at 580 sec (Figure 4.16). Initially, node 1 is primary and serves requests from clients accessing key set 1. At 435 sec it steps down and node 2 takes over as primary. At about the same same time, an additional set of clients starts accessing a different set of keys (key set 2). As node 2 has an empty cache (first time serving as primary), we observe a performance hit of up to 65% (Figure 4.16). We note the reduced throughput per client group in 435-560 sec as they share the replica group. At 560 sec the set of clients accessing key set 1 stop executing, briefly improving throughput for clients accessing key set 2. At 580 sec, node 1 is promoted to primary again. While node 1 has keys in its cache, they are

[3]We use a slightly different testbed (Intel Xeon Bronze 3206 8-core CPU 1.90GHz, 32GB DDR4, Micron 5200 MAX SSD) in this experiment.

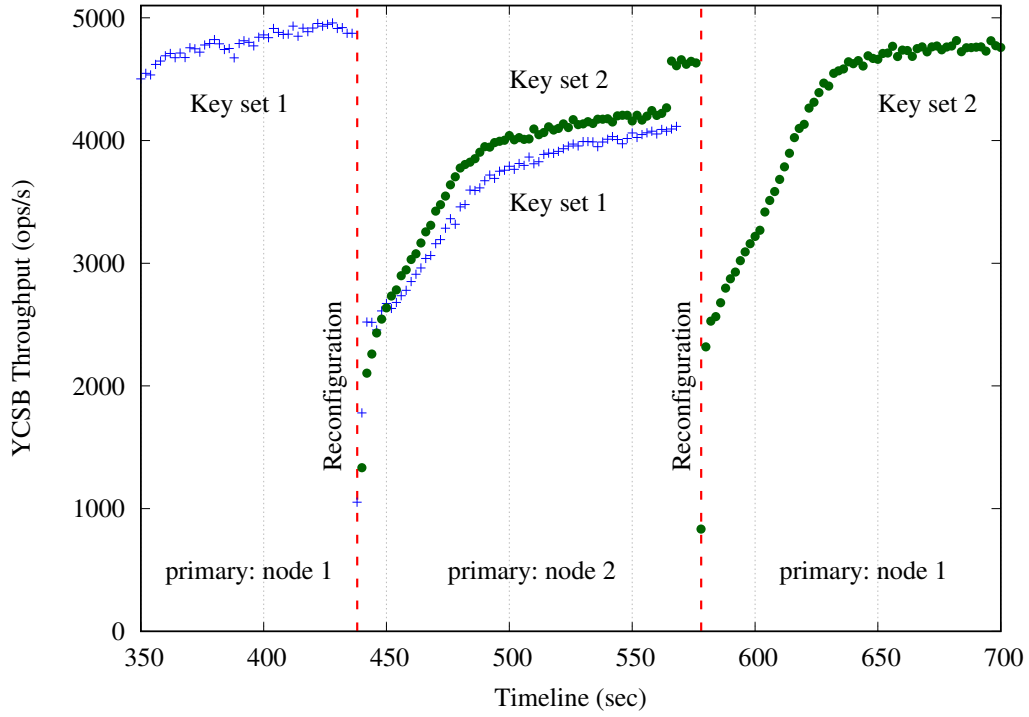


Figure 4.16: Node 1 serves as primary again at 580 sec after a second reconfiguration (RHM disabled, working set changes)

no longer useful as the working set has changed, resulting again to a performance drop. Enabling RHM (not shown in Figure 4.16) results in a smooth transition from node 1 to node 2 (for clients accessing key set 1) and back to node 1 (for clients accessing key set 2).

4.3.7 Performance overhead

In this section we study the performance overhead of our mechanisms, namely the monitoring and logging of read requests into in-memory buffers (the read-hits buffers) and their communication to replica nodes. We focus on the performance impact on throughput under a 100% read workload, in order to fully stress our mechanisms which only apply to read operations.

We repeat the experiment of Section 4.3.1 using SSD as back-end store and we measure the performance overhead in terms of throughput while the system is at steady state

without applying a change of the serving node. In this case the CPU utilization is at 80% (combined user and system time). We break overhead in two parts: (a) the impact of logging every read request and (b) the impact of replicating the read log under different configurations. In our first experiment we enable the logging mechanism but do not replicate the read log. Although we do not replicate the read log across replicas, we still swap the read-log buffer (part of the double buffering being performed, Section 4.2.1) every 90 ms. The average performance impact of logging is found to be 0.6% (average of 10 runs with a relative standard deviation of 0.16%).

We next study the impact of our mechanism with the replication of read log enabled. The reported impact is cumulative, meaning that it includes the cost of logging of read requests and the cost of replication of the read log buffers. We experiment with different replication intervals and number of replicas to sync the read log with. When only one replica gets the read-log updates the performance overhead is 0.9%, increasing to 1.6% when all replicas receive the updates (average of 10 runs for each configuration, with relative standard deviation 0.25% and 0.21% respectively). Experimenting with different replication intervals (90ms and 1000ms) does not seem to have any impact on the overall system performance. A short replication interval creates small frequent batches of read log that are replicated across replicas while a long interval creates less frequent but bursty batches. Overall, we find that the performance impact of our mechanism to maintain up-to-date read caches across replicas is minimal even under stress.

Finally, we focus on the cost of RHM maintenance under resource strain, caused by the sharing of resources between a primary and a backup replica co-located on the same node on AWS EC2. We use a single-shard deployment and focus on the performance (throughput) of a single node hosting the primary replica and a backup. To fully stress the RHM system, we use a read-only workload. We compare unmodified MongoDB to the same system with RHM enabled with a workload that drives the former at an average CPU utilization of 70%, considered fully utilized. At that point, we observe 4.65% lower throughput for MongoDB with RHM vs. the unmodified, as seen by clients. For a lighter loaded server, we observe no noticeable impact on overall system performance. While the impact is considered low even under stress, RHM could be temporarily stopped or selectively apply read

%Hints Applied	Cache miss rate	CPU		Disk reads	
		usr+sys	I/O wait	MB/s	ops/s
10%	39%	5.2%	0.37%	8.8	484
20%	33%	7.3%	2.36%	16.2	794
33%	31%	10.2%	4.26%	25.1	1137
50%	30%	13.4%	5.97%	34.9	1501
100%	30%	24.7%	8.62%	60.9	2426

Table 4.1: Applying fewer read-hints lowers RHM overhead, at the cost of reduced cache efficiency after reconfiguration

hints to reduce it even further.

4.3.8 Selectively applying read hints

As noted in Section 4.2, a secondary replica can selectively apply the read hints it receives via RHM from the primary to reduce the overhead of the mechanism. Here we quantify the overhead (CPU and read I/O) attributed to RHM when applying a fraction of the read hints received (10%, 20%, 33%, 50%) under a YCSB workload with a uniform access pattern. Our results highlight the point that RHM can be applied in an adjustable manner, in line with the amount of resources available to a specific deployment at any point in time. In addition, RHM can be switched off during periods of excessive load, and turned back on after such periods or after more resources are provided to a deployment.

Table 4.1 shows that as the amount of hints applied is reduced, RHM overhead (CPU and read I/O[4]) is also reduced while the cache miss rate on the node that takes over as primary after reconfiguration increases. The cache miss rate on the primary node (which applies all read requests) is 30% at steady state. We also observe that applying 50% of read hints on replicas matches the miss rate on the primary allowing a smooth primary transition between nodes. This is evidence that significant benefits are possible by applying

[4]We use `mpstat` and `iostat` to collect CPU and I/O statistics respectively; the monitored I/O device is used exclusively by MongoDB

even a small fraction (10-33%) of read hints.

Another key point is that the RHM mechanism can be turned off during periods of overload (90-100%). These are not expected to last too long, either because they are typically due to bursts and expected to recede or will be absorbed by adding new resources (elasticity actions), at which point RHM dissemination can resume at a level (% of hints) adjusted to the current level of load.

4.3.9 Space overhead

Read-oplog entries contain just the minimum description of requested data (IDs and some filters) necessary to identify the requested data. To reduce the read log size and the amount of data sent over the wire, we omit (unnecessary to the replicas) information regarding the session id, signature hash and other fields of the read request used on primary node. A replica that receives read log batches can reconstruct and apply the corresponding read requests. Listing 4.1 contains a human-readable representation of a read log entry. In this case the space overhead of the read log entry is 39 bytes (the collection name and the contents of the filter).

```
{
    find: "usertable",
    filter: {
        _id: "user8458018675393720714"
    }
}
```

Listing 4.1: Read-hints buffer entry

As the read log is replicated in batches across replicas, the communication interval affects the size of each batch but not the total data transferred over the network. A replication interval of 1000 ms corresponds to a batch size of 2,260 log entries (matching the serving rate of read requests reported in Section 4.3.1 using SSD as back-end store). A replication interval of 90 ms (default in our prototype) corresponds to a batch size of 205

entries. In both cases, 2,260 requests/sec are communicated to replicas, resulting to 88 Kbytes/sec per replica. The required network bandwidth to replicate the read log is analogous to the serving rate of reads on primary. However we communicate only a minimal description of read requests as hints and not the data itself. In the experiments described in this section, we log all read requests. In Section 4.3.10, we evaluate our optimizations to further reduce the size of the read log.

4.3.10 Optimizations to reduce read-hints buffer size

Batching reads operations provides an opportunity to reduce the total size of the read log to be shipped across replicas as multiple requests for the same data can be logged just once (summarized) in the buffer. This depends on the access pattern of data, the replication interval of read log (as summarization can be performed for requests within the same interval) as well as the serving rate of the system. A system with high serving rate using a long replication interval and featuring strong locality in the access pattern, can greatly reduce the amount of transferred data across replicas.

In this section we study two different access patterns (uniform, Zipf) under two different read log replication interval settings using the fixed serving rate of our system using SSD back-end. Under uniform accesses, almost all the requested keys in every interval are unique, even when the duration of the interval is set to 1,000 ms. Due to the relatively large dataset size (50 million unique keys) it is highly unlikely that during an interval the driver selects the same keys more than once. Under the Zipf distribution, we find that 12% fewer data are transferred to replicas when we use the default read log replication interval of 90 ms. Setting the interval to 1000 ms, 21% fewer read log data are transferred over the network to replicas.

When network bandwidth is scarce, one could apply compression or use more sophisticated deduplication techniques [192] on the read log to further reduce network traffic.

4.3.11 TPC workloads

This section extends our evaluation using workloads modeled after two popular TPC benchmarks, TPC-C (online transaction processing) and TPC-H (online analytical processing).

TPC-C

We experimented with TPC-C [38], a popular online transaction processing benchmark emulating a commerce system with five types of transactions. While initially designed to test traditional RDBMS systems, we used a recent implementation that adapts it to match the NoSQL document data model and to test transactional features [118]

The database is populated to simulate 500 warehouses resulting to 66 GB of data size (41.5 GB on disk space using compression). To have a continuous view of performance, we adapted the benchmark to report throughput and response time for each query every 2 seconds.

Queries are served by the primary node at the start of the run. Initially we aimed to shift the read-serving node to the nearest secondary during the run. However, in this way we could re-target only the `STOCK LEVEL` query (a read-only query) [118] towards the secondary node. We thus moved towards scenarios where we switch the primary node within the replica group. This may occur during a failover action or to improve performance [53, 96, 138, 152]. In this case, all transactions are executed in the node that serves as primary. However, switching the primary yields minimal impact on performance (we observe no cold-cache misses). An analysis showed that this is because most of the queries have a similar pattern (*read, then update (modify) the same keys*), thus eventually propagating the read working set to all replica caches, achieving as a side effect the benefits of our mechanism. We wanted to investigate whether a different type of workload, online analytical processing, has different characteristics. Our results are summarized next.

TPC-H

TPC-H [39] emulates a decision support system or business intelligence database environment tasked with providing answers for business analyses on a dataset, initially designed

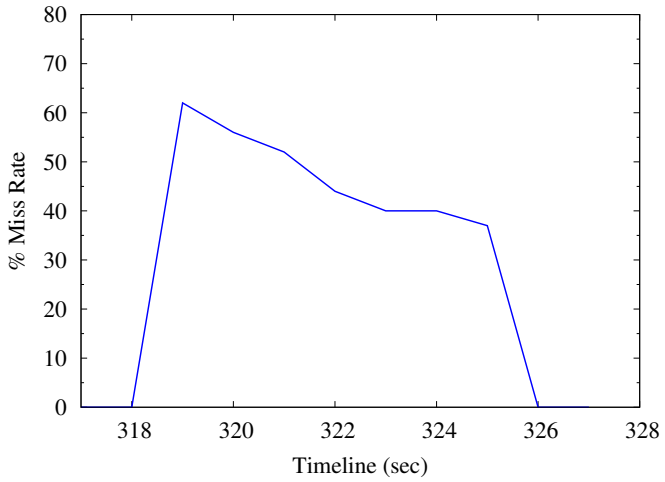


Figure 4.17: Cache miss rate after migration of the analytics query

for traditional RDBMSs. The workload is generally read- and scan-intensive. We implemented a subset of the benchmark[5], namely the "Pricing Summary Report Query (Q1)" compatible with the MongoDB query model. This query reports the amount of business that was billed, shipped, and returned. The query exhibits a high degree of complexity. As it has to scan and fetch data from disk and then perform aggregation operations on them (e.g. sum, average), it is both I/O and CPU intensive. We load the database using the DBGen[6] tool using a scaling factor of 10. To improve transaction performance, we built indexes on fields used for the query. The total database size is 29 GB (resulting in 13 GB on-disk allocated space using compression). The query parameters (e.g. the shipdate interval) are randomly selected.

Initially the query is executed on the primary node with a warm cache (having previously executed another instance of the query), achieving an execution time of 1.2 sec with a nearly 0% cache miss rate. After workload migration (changing read replica) with our mechanism enabled, the query execution time is the same as the new read node has a warm cache. After disabling our mechanism however, replica caches cannot stay in sync with the primary node's cache. Thus, after a migration, query execution time increases to

[5]The database schema could be adapted to better fit the NoSQL document data model but this is out of scope of this work

[6]DBGen is a TPC provided software package that must be used to produce the data used to populate the database.

6.3 sec (a 5.25 times slower query execution time) due to initial cold-cache misses. Figure 4.17 shows that the cache miss rate is between 60% and 40% during the execution of the query that performs read and scan operations. Our mechanism is able to maintain warm caches across replicas, achieving an 80% reduction of execution time.

4.4 Summary

In this chapter we propose a low-overhead mechanism for maintaining warm read caches across all replicas in systems that serve reads from a small number of (typically one) replicas. We find that the mechanism can have significant performance benefits during reconfiguration actions, avoiding large performance drops for long periods of time (order of minutes). The performance drops avoided by our mechanism are observable even with faster storage devices and are proportional to the size of the cache. Mixed (read/write) workloads featuring non-overlapping read and write working sets are also exposed to the problem described in this chapter and benefit from our solution. All in all, the proposed mechanism is low cost, easy to implement and retrofit in existing systems, and results in significant benefits during reconfiguration actions. Its low performance impact observed under periods of resource strain can be avoided by reducing or stopping the maintenance of read hints during such periods. While the benefits of our mechanism are not on the common path of system performance, the challenge addressed in this work has the potential to be even more impactful in the future, as reconfigurations become more frequent for management actions such as replica rejuvenation, proactive recovery, adaptive placement, and to mask the performance impact of resource-heavy activities.

Chapter 5

Amoeba: Aligning Stream Processing Operators with Externally-Managed State

In multitier service architectures the application building blocks are separate layers. The application logic is often separated from the data persistence layer (storage system). Distributed middleware systems rely on data stores to maintain and manage their state. Stateful services generate their response by executing their processing logic on its state persisted on an external data storage system. As these systems usually have strict performance requirements, the underlying data stores need to deliver its functionality with even tighter bounds. Tuning both systems is a challenging task. In addition, the cross-system communication for the data access path can critically affect the overall service performance. In this chapter we study the performance improvements of more efficient cross-system communication by aligning data stores with distributed middleware platforms

We use stream-processing systems (SPSs) as a distributed middleware platform use case. Scalable stream-processing systems and analytics are key to a number of high data-volume Internet services [7, 21, 50, 75, 115, 160, 176]. As real-time data analytics provide valuable insights, there are cases where stream-processing platforms process trillions of events per day [21]. A stream application can be modeled as a directed dataflow graph

comprising processing operators [47]. Each operator transforms data flowing from its inputs to its outputs. Stateful operators preserve a sequence of recent data as ephemeral internal state and apply a transformation on a subset of its incoming data (e.g. aggregating values over a time window). Storing intermediate (ephemeral) state of streaming applications in a reliable, recoverable manner is important for the fault-tolerance of long-running jobs.

Early SPSs stored ephemeral operator state in memory or on an external data store (e.g., an SQL server). Modern systems offer data management mechanisms using embedded local key-value stores (KVS), typically coupled with periodic checkpoints stored on external storage systems [67, 144] for fault-tolerance. Other systems externalize data management to distributed storage systems for out-of-the-box fault-tolerance, recovery, and elasticity [46, 95].

Persistence of *external state* of streaming jobs is a separate but equally important concern. External state refers to data produced by streaming jobs or by other applications or systems [50,75,115,176], or ground-truth table data that can be accessed during the execution of streaming applications. External state has typically a lifecycle that is disconnected from that of streaming applications and is often stored on scalable KVSs [78, 115, 176]. Current state-of-the-art solutions, such as IBM InfoSphere Streams, provide support for general-purpose access to external state on scalable KVSs [160, 164]. However, lack of coordination in the placement/partitioning policies of SPSs and external-state KVSs, results in unnecessary network cross-talk (often on the critical path of a streaming application), and mis-alignment of adaptation policies such as elasticity.

In this thesis, we build upon the idea that the performance of accessing external state can be improved via co-location of internal and externally-managed state partitions. This is possible either via appropriate (informed) initial placement of such partitions and alignment of SPS and KVS key partitioning schemes or via migration of tasks (along with their internal state) or externally-managed state partitions, with the ensuing data movement where appropriate. Early experimental results (poster [94]) report the benefits of aligning data and task placement/partition policies in scalable SPSs with externally-managed state KVSs. That study used a simple application derived from the Yahoo streaming bench-

mark [78] deployed on the Flink [67] SPS using Redis as an external-state KVS. The results provided early evidence of significant performance benefits on manually created configurations and small-scale setups.

In this work we extend those early results in a number of directions. We describe the design and implementation of an actual system that we call Amoeba[1] that implements the alignment between SPS task partitions and externally-managed KVS-based state. We experimentally evaluate Amoeba on an advanced, complex streaming benchmark application, Linear Road [52, 180], on large-scale deployments of up to 64 AWS nodes. Our work makes the following research contributions:

1. A broad investigation of the benefits of automatically aligning SPS tasks with externally-managed state in large-scale production-quality workloads.

2. Determination of the benefits possible with shared access on externally-managed tables, and how these benefits change with increasing system scale.

3. Investigation of combined actions (shipping data to tasks or vice versa, adaptation of partitioning schemes) to align externally-managed state partitions with SPS task partitions to exploit locality.

4. Development and use of a scalable implementation of the state-of-the-art Linear Road benchmark [52, 154, 180].

The research work reported in this chapter provides a novel solution to the research problem of automatically streamlining and integrating state management capabilities across stream and storage technologies, a complex undertaking that today requires deep human expertise, making it a hard and failure-prone undertaking.

The rest of this chapter is structured as follows. In Section 5.1 we position Amoeba in relation to previous research in this space. In Section 5.2 we detail our design and prototype implementation of Amoeba. In Section 5.3 we describe our implementation of Linear

[1] An amoeba is a type of cell or unicellular organism that has the ability to alter its shape, primarily by extending and retracting pseudopods [4]. Its adaptivity as a defining characteristic inspired us to name the system presented in this chapter after it.

Road for the Flink SPS using the Redis KVS for storing externally-managed state. In Section 5.4 we discuss our extensive evaluation of Amoeba on AWS deployments of up to 64 nodes. Finally, in Section 5.5 we conclude.

5.1 Related work

Stream processing applications can be expressed as a collection of processing tasks (operators) connected in a dataflow graph [47, 178], with each operator transforming data flowing from its inputs to its outputs. Operators can be *stateless* or *stateful*. The output of stateless operators (such as filters) is based on the current inputs only and does not take into account state from previous computations. Stateful operators (such as aggregations) build state from past inputs and use it to influence the processing of future input.

Amoeba addresses the alignment between tasks managing stream-processing state and partitions of externally-stored and managed state. Here we review related research in three areas: SPS access to externally-stored and managed state (§5.1.1); the management of SPS internal state (ephemeral operators state) (§5.1.2); and finally, task placement/alignment to improve performance in different contexts (§5.1.3).

5.1.1 Storing and accessing external data

Streaming applications often require access to *external data* on the common path, typically stored on external systems whose lifecycle is decoupled from that of the stream-processing application. Access to external data may be needed in the context (and for the purposes) of either stateful or stateless operators. Management of external state is orthogonal to the management performed by SPS of internal state (ephemeral operator state); the latter is the focus of Section 5.1.2.

Several production use-cases highlight the need to use external storage systems to share state across different processing applications and management systems [50, 75, 164]. Existing scalable SPSs provide ways to interact with scalable external stores. One way is through source/sink operators channeling streams to scalable key-value stores such as

Redis [67, 144] and through user-defined operators with custom code accessing external state via publicly-available KVS client drivers and APIs. Another way is to extend streaming frameworks with general-purpose put/get access to remote stores [164] as supported by IBM InfoSphere Streams *distributed process store*, and demonstrated in commercial use cases [160]. This model also allows to combine "data-at-rest" (batch processing) and "data-in-motion" (stream processing) analysis.

Accessing external state *in the common path of stream processing* is known to be a cause of overhead (higher latency, as well as protocol overhead). With out-of-core datasets stored as external state (e.g., user profiles in large Internet services [75, 144, 176] or user activity [50]), external state is a performance pain point that is not well addressed so far. Amoeba addresses this challenge through the alignment between tasks managing stream-processing state and partitions of externally-stored and managed state.

5.1.2 Storage solutions for internal operator state

Stateful operators build state from past inputs. This internal state is ephemeral and is tightly coupled with the lifecycle of the operator. Modern SPSs follow two directions to manage the operators internal state: using an embedded data management system such as RocksDB [67,144], or storing it on external data stores [46,131,139]. An embedded store results in efficient local data access, while an external data store has the additional benefit of independent failure modes, although at a cost of higher access latency.

A challenge with internally-managed state is the need to re-organize operator state via network transfer across nodes during reconfigurations of the SPS, which has led to recent research in reducing the impact of such reconfigurations [87,112,191]. Using a distributed storage system to persist internal operator state has the benefit of allowing systems to quickly recover from failures and to efficiently reallocate stateful operators across several newly partitioned operators to provide scale out. With certain SPS using external storage systems for reliability and reconfiguration (elasticity), each processing operator creates periodic checkpoints that are mirrored to an external distributed storage system (e.g. HDFS). In case of failure or reconfiguration, the stream processing system can recover its state

from the external store [67, 87, 144, 191]. The external store is *not involved in the common path of stream processing* and is only used during the checkpointing or recovery process.

Exposing internal operator state and allowing users to query the state of the streaming job from the outside has recently been introduced in the Flink SPS via support for *querable state* [14]. However, this approach applies to internal job state only and does not extend to external data whose lifecycle is disconnected from that of a streaming application.

Combining distributed state management solutions to support both external state and internal operator state via a single platform in modern stream processing frameworks could lead to efficiency and versatility benefits. However, determining how to effectively combine the two types of systems remains a challenging research problem, as has also been pointed elsewhere [181].

5.1.3 Placement/alignment of SPS tasks

Previous work studied improvements in the placement of processing tasks to reduce the communication overheads between operators, migration of operators and replication [93, 149, 158] or study the checkpoint placement policies [87, 162]. Controlling the placement of externally-managed state to align with streaming tasks for lower latency, a focus of this work, has not been made so far. Aligning data and processing partitions on a common topology to improve data locality is a challenging task. Work on modeling load balancing, operator instance co-locations and horizontal scaling as one integrated optimization problem that can be solved using mixed-integer linear programming (MILP) [141] are related to Amoeba in that it pursues opportunities for co-location (in this case, operator instances) to improve system performance. Previous research looked into task assignment for state sharing [190] and on shared store [139] to support this. This work as well as previous research on improving SPS state partitioning schemes for load balancing [97, 123, 143, 165] are orthogonal to our approach.

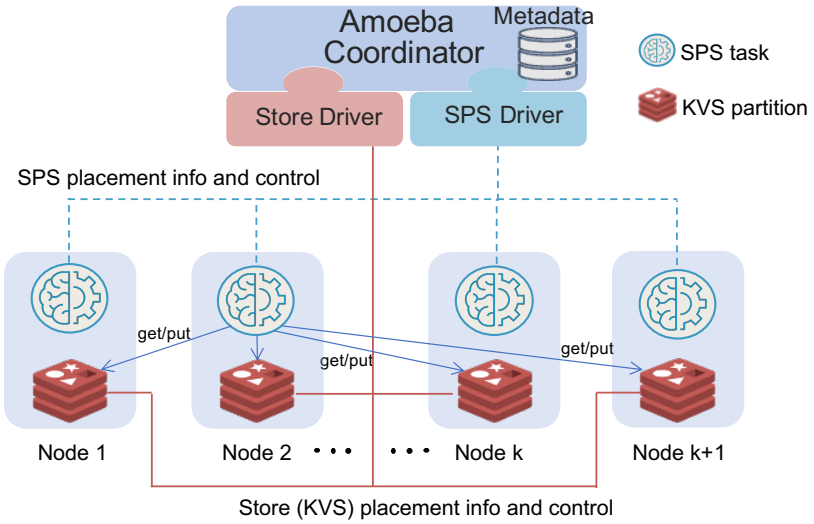


Figure 5.1: Amoeba system design. Without any alignment, each SPS task will be accessing keys from all KVS partitions.

5.2 Design and implementation

In this section we describe the design and implementation of Amoeba, a system for adaptive and dynamic alignment of partitioning schemes and assignment of task and data partitions to nodes to improve system performance by minimizing data access over the network.

Figure 5.1 shows the Amoeba system design. It functions as a typical autonomic monitor-analyze-plan-execute (MAPE) loop [77] whose main purpose is to monitor and coordinate SPS-KVS systems, aiming to achieve efficient SPS access to external state. Amoeba follows a modular design approach using SPS and data store specific drivers as plugins to communicate with the underlying systems. Drivers are responsible to discover metadata and to realize (effect) the aligment plan on the underlying systems.

At the core is the Amoeba **Coordinator**, whose role is to monitor and control interdependent SPS and KVS platforms and their relationships, through the driver modules that encapsulate domain-specific knowledge. The Coordinator incorporates a metadata repository responsible for storing information, such as application descriptions, deployment topologies of both systems. It periodically contacts SPS and KVS to discover topology

and partition related information (e.g. partition assignment across instances). The coordinator correlates the metadata of both systems to create an alignment plan by a combination of task/partition migrations and modifications to the partitioning scheme on SPS and/or KVS, as applicable (§5.2.2). The design of the Coordinator is agnostic to the underlying SPS and KVS.

An **SPS Driver** is a module responsible for interaction with an SPS, incorporating domain-specific knowledge about discovering information and controlling each SPS via external management APIs. An SPS Driver discovers streaming application descriptions, physical deployment topologies (assignment of operator partitions to SPS worker tasks), and how streaming operators are routing their inputs into partitions of downstream operators. Operators partition their outputs when their downstream operators have multiple instances, assigning a non-overlapping set of their output (partition) to each instance of each downstream operator. The partitioning scheme is usually based on hashing, although it can also be based on simpler round-robin key distribution approaches or more complex load balancing methods. Following an alignment plan decided by the coordinator, an SPS may adapt its partitioning scheme and assign partitions across processing tasks and/or reconfigure to execute task-migration actions.

A **Store Driver** is a module responsible for the interactions with a *data store* (in principle any scalable storage system, although scalable KVS are the prevailing technology today), focusing on data partitioning, the assignment of one or more partitions across data-store instances, and partition migrations. Amoeba plans its actions upon the understanding that data stores may adapt similarly to SPS tasks, i.e. by adapting their partitioning scheme, re-assign partitions across its instances, as well as via data migration actions.

Naturally, some actions have a higher impact than others, and there may also be constraints on what actions are feasible given the configuration of resources on the field. Amoeba takes these factors into account in proposing actions. Next we focus on details of Amoeba alignment plans for improving the efficiency of SPS access to external state.

5.2.1 Amoeba inputs and metadata discovery

Amoeba requires users to provide handles and credentials (access rights) to management components and APIs of the underlying systems (SPS and KVS) as well as monitoring endpoints. Amoeba discovers (rather than require a human expert to provide) the deployment topology of the SPS and KVS and the execution graph of streaming applications. Through static analysis of the application's code or deployment package, Amoeba may pinpoint the operators accessing external state. This information is augmented and cross-checked by dynamic monitoring of data-access requests. For completeness, Amoeba allows application owners to augment this information by identifying operators (names) that access external data along the associated external data tables. The Amoeba coordinator periodically communicates with the SPS and data store(s) to draw the above metadata and to discover opportunities for alignment to improve data locality.

5.2.2 Planning alignment actions

An alignment plan is a collection of alignment actions that aim to improve locality of one or more operators. Amoeba generates alignment plans in three ways, exemplified in Figure 5.2, where a processing operator with 3 parallel instances accesses an external data table split into 2 partitions (shards) (Figure 5.2(a)). Amoeba may adjust the parallelism of either the SPS or the KVS via elasticity actions to improve the potential for alignment between the two systems; in Fig. 5.2(b) it creates an additional KVS server on Node 3. Amoeba may next migrate processing tasks or data partitions, leading to a deployment that increases co-location between the operators and the corresponding data tables; in Fig. 5.2(c) it migrates a task from Node 4 to Node 1. Finally, Amoeba improves data locality by aligning the partitioning schemes of the two systems to exploit the co-location between processing operators and data tables on Nodes 1-3. Amoeba considers known global constraints (e.g. data tables not allowed to move to or out of specific nodes) and resource limitations (e.g. overloaded nodes that cannot host more processing operators or data shards) in its decision-making process.

Next we describe the process of alignment plan generation and the decisions involved

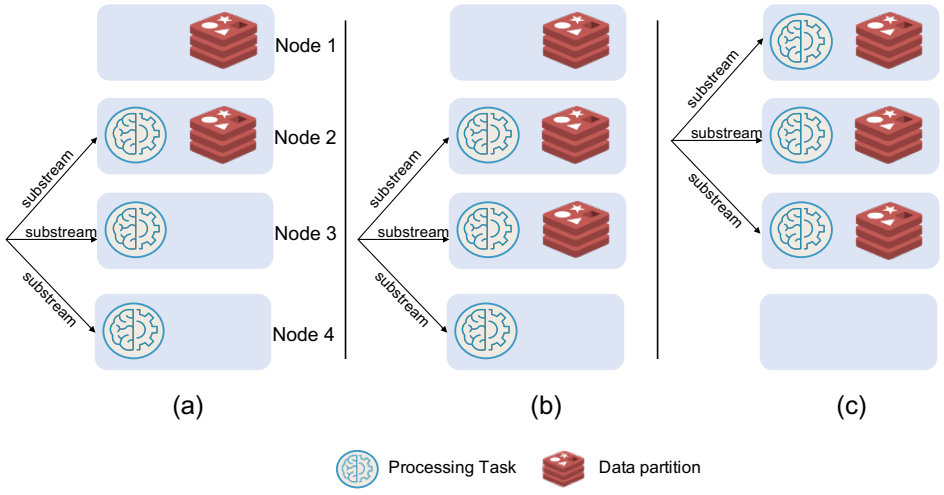


Figure 5.2: Example: (a) Initial deployment, (b) creation of new shard, (c) task migration and partition-scheme change

in more detail (Figure 5.3).

Adjust the parallelism of SPS or KVS: A difference in the degree of parallelism between the SPS and KVS deployments (as in Figure 5.2(a)) will prohibit full co-location between the two systems, even after migration and partitioning-scheme changes (described next). Where possible, Amoeba considers adjusting the parallelism (via elasticity actions) on either the SPS or KVS as a way to achieve optimal alignment of the systems. Even when SPS and KVS may have had the same degree of parallelism at some point in time, this may be disrupted by an externally triggered elasticity action on one of the two systems (e.g. the SPS, due to a load surge), without a corresponding action by the other. Amoeba can detect this and trigger a corresponding elasticity action (when feasible) to realign both systems. Amoeba prefers scaling up the system with the lower parallelism, as scaling down may negatively impact overall performance. As elasticity actions may be subject to cross-check with other systems (such as external elasticity controllers) and management controls (for billing, etc.), the Amoeba decision process may involve a human operator to explicitly approve actions.

In Fig.5.2(b), Amoeba adds a new KVS instance on Node 3 so that processing tasks and data partitions have the same parallelism, and re-balances the KVS. Then, it explores

potential for more co-location through data or task migrations:

Migrate tasks or data to increase co-location: Where processing operators and data tables are hosted on separate nodes (as in Nodes 1 and 4 in Fig. 5.2(b)), Amoeba will consider migration actions to increase the co-location. Available options of migrating SPS partitions or KVS data partitions (shards) are taken into account in deciding the appropriate adaptation action. Amoeba selects either SPS or KVS migration depending on the availability of online, low-impact migration mechanisms [87, 112, 151, 191] in either system. Amoeba assumes apriori knowledge of whether such mechanisms are supported in a SPS or KVS in its metadata catalog. If both support such mechanisms, it defaults to the SPS. Sometimes there are constraints that do not permit such actions (e.g. no access rights or limited processing capacity on certain nodes). Amoeba takes into account constraints, rules, and policies in creating migration plans. If migrating a KVS entirely off of its original deployment is not an option, creating new data replicas on the SPS nodes (rather than effecting a full migration there) may be the best possible middle ground.

In Fig. 5.2(b), Amoeba moves the processing task initially placed on Node 4 to Node 1. Next, it aligns the partitioning scheme of the systems to fully exploit data locality:

Align SPS/KVS partitioning schemes: In any deployment topology, a key Amoeba step is to align the partitioning scheme of the systems to improve data locality. Where it finds that data partitions and processing tasks are already co-located on certain nodes, Amoeba can trigger changes to the partitioning scheme to improve alignment, i.e. the same stream and data keys be routed to the same node. This alignment can be realized on the partitioning scheme of either the SPS or the KVS. Although it is feasible to dynamically repartition a scalable KVS [99], it is usually considered a heavyweight task. In addition changes on the partitioning scheme of the external database may also impact other systems or applications that access the external data. Thus partition-scheme change on the SPS is typically preferable. In Fig. 5.2(c), Amoeba detects co-locations on Nodes 1-3 and changes the SPS partitioning scheme to assigns keys in each KVS shard to the SPS task hosted in the same node.

Through continuous monitoring, Amoeba analyzes the performance impact of its alignment plans and may re-evaluate if needed (e.g. in case there are still remote data accesses),

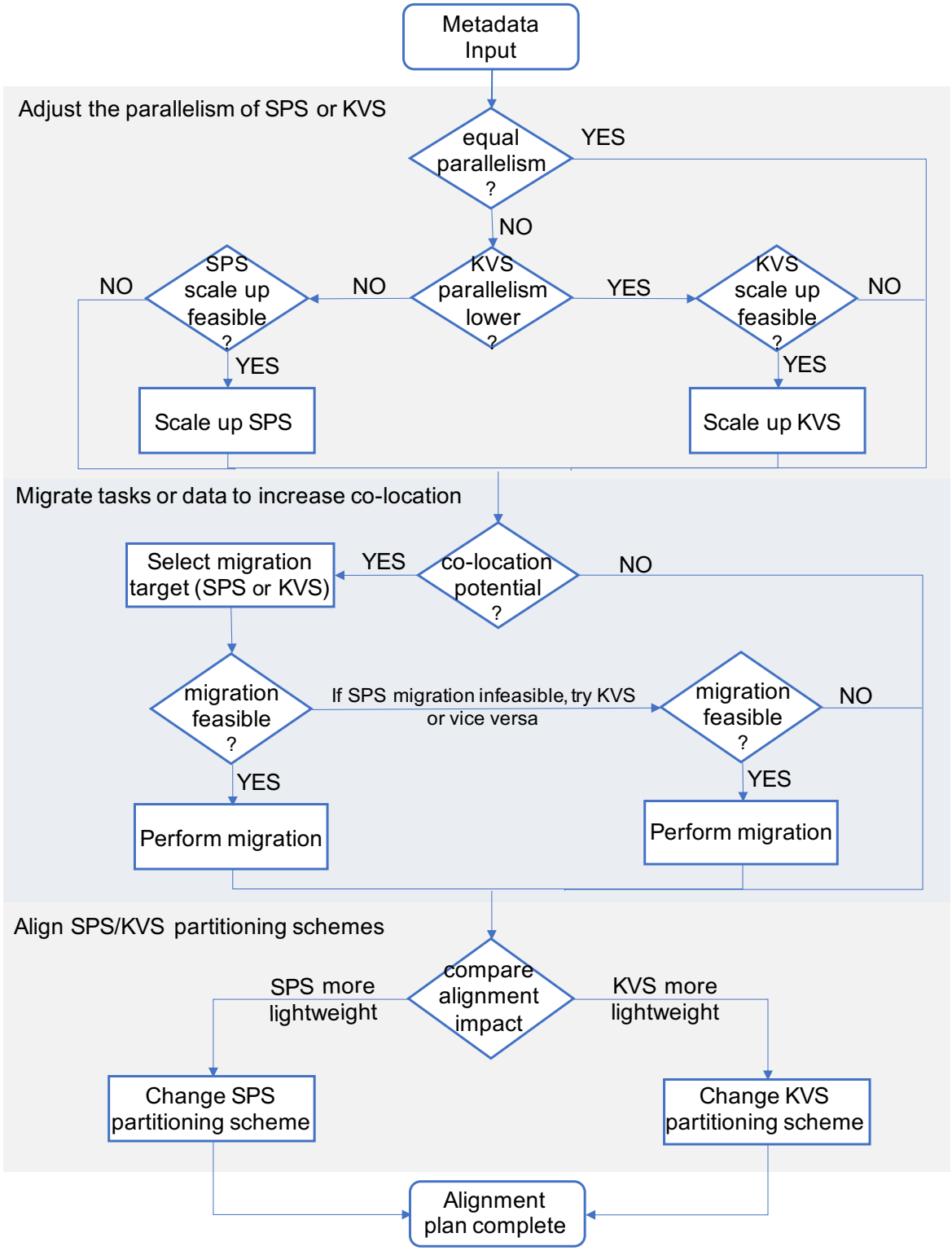


Figure 5.3: Amoeba decision-making process

similar to previous approaches in online optimization of allocation decisions [51].

Figure 5.3 depicts the Amoeba decision-making process incorporating its three key mechanisms along with decision points that check on the impact of actions or constraints. As an alignment of SPS-KVS partitioning schemes is needed after either a migration or an elasticity action, the decision-making process starts by considering elasticity first, followed by migration, and ends with partition-scheme alignment.

In our evaluation we demonstrate the decision-making process of all three key alignment mechanisms, namely elasticity actions to adjust parallelism (§5.4.5), data migrations (§5.4.4) and partitioning scheme alignment (§5.4.2 - §5.4.3)

5.2.3 Prototype implementation specifics

Our prototype implementation of Amoeba includes a Coordinator and drivers for the Flink SPS and Redis Cluster KVS. The Coordinator incorporates a scheduler that periodically invokes both drivers to discover changes in the deployment topologies. The Flink SPS Driver contacts the Flink Job Manager and extracts the necessary metadata for the running jobs (streaming applications), including details on operators and their accessed state, deployment parallelism, partition placement, etc. The Redis Store Driver is based on the *lettuce* Redis client library[2]. This Redis driver contacts any Redis nodes to get metadata information regarding partition assignment across Redis instances. Both Flink and Redis drivers require no modifications to the underlying platforms.

The Flink and Redis drivers are also used to collect monitoring data exposed by the underlying platforms. In Flink we use the REST API to access the monitoring data exposed by the JobManager. We also enriched the Lettuce Redis client library to expose the Redis requests distribution across nodes to the JobManager metrics. Monitoring is performed via periodic sampling and thus has low overhead. Amoeba can access the monitoring data through its Flink driver. An alternative would be to get the access pattern of Redis requests from the Redis servers directly using the *monitor* command[3].

Elasticity and migration actions. Amoeba discovers nodes joining or leaving the clus-

[2]https://lettuce.io (retrieved September 2021)
[3]https://redis.io/commands/MONITOR (retrieved September 2021)

ter on the SPS and can trigger a corresponding KVS action, a Redis *cluster rebalance*. Redis splits data horizontally to 16384 partitions, also called *shards* or *slots*, mapped to Redis node instances. Our implementation supports Redis cluster rebalance to a different set of instances by re-assigning its 16384 partition slots across the available Redis instances. Amoeba effects elasticity on Flink via its standard mechanism based on savepoints [15].

Data migration is supported by Amoeba via the creation of new replicas and change of leader within each Redis replica group, as a means to migrate KVS partitions to SPS nodes. This is one of the options considered when KVS nodes do not have processing capacity (or are otherwise constrained) to host SPS processing tasks. Redis supports the leader-follower (master-slave) replication scheme. Every partition has a master replica that serves all read and write requests and a configurable number of follower replicas that asynchronously get updated by the leader replica. The Amoeba Redis driver supports data migration by creating a new follower replica on an appropriate SPS node. When the new replica catches up with the primary, the driver reconfigures the group, promoting that replica to primary, allowing for local reads/writes between co-located SPS tasks and KVS partitions.

Partition-scheme alignment. An important mechanism exercised by Amoeba is changing the partitioning scheme in either the SPS or the KVS platform (or both) so as to achieve alignment, namely that external state and operator key routing (and internal state, in case of stateful operators) are partitioned the same way. Flink uses the *KeyBy* transformation, which logically partitions a stream into disjoint partitions based on some field(s) of the tuples (called partition key) [13]. All records with the same key are assigned to the same partition. Internally, Flink KeyBy is based on the MurmurHash algorithm[4]. Since KeyBy does not allow customization, in our prototype we use the Flink *Partitioner* API interface[5] that allows users to implement their own partitioning method. We have implemented a remote communication API to the Flink Partitioner API through which the Coordinator dynamically adapts the partitioning scheme of certain operators according to its alignment plan. The communication between the Coordinator and Flink partitioner is carried

[4] `https://en.wikipedia.org/wiki/MurmurHash` (retrieved September 2021)
[5] `https://ci.apache.org/projects/flink/flink-docs-master/api/java/org/apache/flink/api/common/functions/Partitioner.html` (retrieved September 2021)

over a sockets-based protocol performing data exchange in a simple JSON format. Redis uses a hash-based sharding scheme based on CRC-32 to distribute data across multiple instances across nodes [33]. The Redis partitioning method differs from the native Flink partitioning scheme. Although Amoeba could make use of database repartitioning as another adaptation mechanism, we have not implemented it in this context. However, ways to dynamically repartition a scalable KVS have been studied in the past [99].

Our prototype offering the above described functionality can support a wide range of experimental scenarios (§5.4).

5.3 Linear Road

To support our evaluation of Amoeba under a complex stream-processing application, we selected Linear Road [52], an extensively used performance evaluation benchmark [180]. Linear Road simulates a tolling system on expressways of a metropolitan area based on variable pricing, i.e., tolls calculated based on dynamic factors such as traffic congestion and accident proximity. Linear Road processes position reports emitted periodically by every vehicle containing its position and speed on an expressway. Besides continuously-evaluated toll pricing and accident detection, the benchmark is also designed to answer historical queries (e.g., account balance and estimations of travel time) that are issued less often.

While Linear Road has been implemented in the past [52, 115], no existing implementation satisfied all requirements we had in this work, namely: (1) Implementation and the underlying platforms to be available as open-source; (2) able to store and manage shared tables in an external database or a key-value store; (3) fully leveraging stream-processing capabilities of the underlying platform (in our case, Flink), e.g., using sliding window operators where appropriate rather than ad-hoc management of internal state; (4) be as faithful as possible to the original benchmark specification [52]. Since no existing implementation available to us satisfied all requirements, we decided to develop our own [154]. In the process this helped us better understand the benchmark's internals and to better relate to and explain our results. In Section 5.3.2 we describe Linear Road's use of internal and

externally-managed state, namely ephemeral operators state (e.g. window buffers) that are used to produce the application output vs. state of broader interest (such as historical table data) even beyond the streaming job that should be accessible outside of the SPS.

We also implemented our own data generator that connects to a scalable data ingestion service (Kafka) eliminating the need for a separate data driver that reads data files and delivers it to the SPS. Our generator also supports additional configuration options, such as number of expressways, duration of the simulation, and number of vehicles in every expressway. As we want to produce a high volume of simulation data and stress-test the SPS, our data generator ensures that there are always 1000 vehicles per expressway by default, reporting their position every second (new vehicles enter a random segment when a vehicle leaves the metropolitan area). We plan to make our Linear Road implementation and data generator available as open source to the community.

5.3.1 Linear Road dataflow graph

In this section we describe our implementation of Linear Road on the Flink SPS, following published benchmark specifications [52]. We first implemented a data generator following the specification of the traffic simulator that is able to produce synthetic data and load them into Kafka. Flink applications are event-time aware, i.e., each element in the stream needs to have its event timestamp assigned. In our implementation we extract the simulation time field of each event and use Flink's *TimestampAssigner* and *WatermarkStrategy* API to mark tuples in the stream [12].

Figure 5.4 depicts the application as a dataflow graph. Operators are annotated with their type (filter, map, window, etc.) and a name that describes their functionality. Several operators access external state on an externally-managed store (KVS) as shown on the right of Figure 5.4 (in Section 5.3.2 we describe the state management needs of the application). The application ingests events from a Kafka topic and deserializes the raw data into structured event objects through a FlatMap operator (named *deserialize*). Next the stream splits into two substreams based on the event type, position report (event Type 0) or historical query (event Type 1). In what follows we describe the Linear Road application

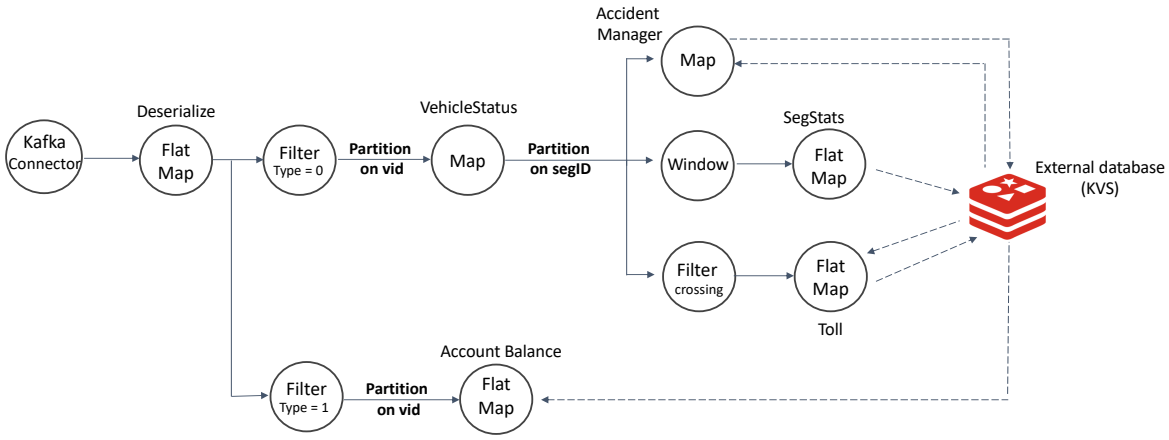


Figure 5.4: Linear Road dataflow graph

through the workings of its key operators: VehicleStatus, AccidentManager, SegStats, Toll, and AccountBalance.

VehicleStatus operator: The position report events sub-stream (upper sub-graph of Figure 5.4) accounts for 99% of all generated events. The stream is partitioned based on the vehicle ID (vid) that emitted the position report, and following that, the *VehicleStatus* operator identifies the status of the vehicle: moving, stopped, or crossing segment. A vehicle is considered as stopped if four consecutive position reports indicate exactly the same position identified by the triplet (xway, segment, position within the segment). To identify the status of a vehicle, the VehicleStatus operator maintains in its internal state the previous position of the vehicle. The internal state of the operator is maintained using Flink's embedded state management system as the lifecycle of the state is coupled with the operator's lifecycle (ephemeral state) and there is no need for sharing that state with other operators (more details on state management in §5.3.2). The operator marks the corresponding fields in the tuple accordingly and forwards it to its downstream operators. The output stream of the VehicleStatus operator is partitioned based on the segment ID (segID). VehicleStatus has three downstream operators consuming its output.

AccidentManager operator: The *AccidentManager* operator discovers accidents that occurred or cleared in the segment. An accident occurs when two stopped vehicles emit reports at the same position at the same time. For every stopped vehicle, the operator ac-

cesses the external database to check whether there is already another stopped vehicle in the segment and reports a new accident. If this is the first stopped vehicle of the segment, the operator updates the database accordingly. For every vehicle marked as "started moving", i.e. a vehicle that previously had been marked as stopped, now reports a new position indicating it has started moving, the AccidentManager updates the external database in order to clear the accident involving the vehicle.

SegStats operator: The application updates the statistics of each segment of the metropolitan area. The statistics include the number of unique vehicles and the average speed of the moving vehicles in the segment in the last 5 minutes. The statistics are updated on a per-minute basis and are used to determine toll charges according to the variable tolling system. We use a 5-minutes sliding window that slides every 60 seconds emitting its output to the downstream operator, *SegStats* to update the external KVS. We use the Flink API to build a custom process window function for the computation. Process functions in Flink come at the cost of a surge in performance and resource consumption when the window closes. The reason behind this is that the elements cannot be incrementally aggregated. Instead tuples are stored in the window's internal buffers as they arrive until the window is considered ready for processing (i.e. when the window trigger fires every 60 seconds).

To avoid a spike in resource consumption when the window closes, we combine the process function with a custom aggregate function that incrementally aggregates elements as they arrive in the window, using the corresponding API. However, the aggregate functions do not provide information about the window start and end times, which is needed to update the external database with the statistics. Combining the process and aggregation functions allows us to have incremental computation and access to metadata about the window as the process function will be provided with the aggregated result when the window closes.

Toll operator: For position reports marked as *crossing*, i.e. the position report indicates that the vehicle enters a new segment, the *Toll* operator dynamically calculates toll charges based on segment statistics and proximate accidents according to the variable tolling system [52]. The operator gets the information by accessing the external database. Finally the Toll operator notifies vehicles for the charges and updates the account balance

of the vehicle in the database.

AccountBalance operator: Type 1 queries are events indicating that a vehicle requests its account balance. The corresponding account balance operator accesses the database and notifies the vehicle accordingly (bottom of Figure 5.4). Similar to the Aurora implementation of Linear Road [52], we did not implement travel-time estimation queries and input for this type of query is ignored.

5.3.2 Linear Road state

Linear Road processes continuous queries and historical queries. According to the specification of the benchmark [52], historic data are usually loaded offline into a storage facility of choice. In processing a historical query request, the system reports an account balance, a total of all assessed tolls on a given expressway on a given day, or an estimated travel time and cost for a journey on an expressway. Historic data should naturally be stored in an external store, accessible and able to be updated by the SPS. The external data store being a different system from the SPS also allows data to have *a lifecycle that is decoupled from that of the stream-processing job*, namely historical data could predate the processing job and should be available after the end of it. In addition, the external store allows sharing of information across multiple applications/systems. This is a common practice in production environments where multiple systems involved in real-time data processing including SPSs and data stores (§5.1). The lifecycle of such data should not be tightly coupled with the lifecycle of a single processing task or job.

The variable tolling system simulated in Linear Road dynamically calculates toll charges based on the current segment statistics (e.g. number of vehicles, average speed of moving vehicles, accidents etc.). Statistics are generated by a window operator as described in Section 5.3.1. Usually such statistics provide useful insights about the metropolitan area and are often used by multiple systems and applications (e.g. reporting and visualization tools). In our Linear Road implementation we use an external data store to support the sharing and exposure of useful information (such as statistics) across operators and multiple systems. Similar design principles were applied by Jain *et al.* [115] and others [160].

Redis table	Updated by	Read by
Segment stats	AccidentManager, SegStats	Toll
Account balance	Toll	AccountBalance

Table 5.1: Linear Road external state and its consumers

For operator state that is not shared across operators or between processing jobs and whose lifecycle is coupled to the lifecycle of the operator, there is no need to use an external store (e.g., operator *VehicleStatus* described in Section 5.3.1).

In our implementation of Linear Road we use Redis as an external key-value store. After breaking down and analyzing the semantics of Linear Road state we decided to use two separate Redis tables (summarized in Table 5.1). The first table stores *segment statistics*, such as number of vehicles, average velocity, and the occurrence of accidents. This table is updated periodically by two different operators, *Accident Manager* and *SegStats* and is read by the *Toll* operator, which calculates the dynamic toll charges. Operator instances processing the same substream partitions should be placed on the same node to maximize the benefit of alignment. Flink does this by default in our deployments.

The second Redis table stores the *account balance of each vehicle*. This structure is updated by the *Toll* operator and used by the *Account Balance* operator to answer the corresponding query. These operators consume different substreams, partitioned by different keys. The corresponding operator instances may be placed on different nodes by the SPS. This is the case where multiple operators that access the same data partition are placed on different physical hosts. In this case, Amoeba opts to co-locate the table with the operator that performs the most frequent access to it, determined based on monitoring data (in this case, the *Toll* operator).

5.4 Evaluation

Our evaluation was carried out on the Amazon EC2 cloud. The resources used are summarized in Table 5.2, with all VMs allocated in AWS region us-east-1. Our Linear Road application ingests data from a 3-node distributed Kafka service deployed on 3 r5ad.large

	#VMs	VM type	#vCores	DRAM	SSD
Kafka cluster	3	r5ad.large	2	16GB	75GB
Flink Job Mgr	1	c5d.large	2	4GB	50GB
Flink Task Mgr	1-64	c5d.xlarge	4	8GB	100GB

Table 5.2: AWS VM types used in our evaluation

VMs. These nodes also host our data generator. The Kafka topic which Linear Road subscribes to is configured with 64 partitions. The Flink cluster deployment consists of a Flink Job Manager, hosted on a c5d.large EC2 server and up to 64 c5d.xlarge nodes hosting Flink task managers. Each node used for the task managers also hosts a Redis instance, part of the Redis cluster. All selected nodes support up to 10 Gbps of network bandwidth.

In this section we evaluate the performance improvements in various Linear Road deployments and scenarios achieved by Amoeba adaptation actions. In particular, in Section 5.4.1 we focus on the impact of unaligned SPS-KVS platforms and demonstrate that the throughput *per node* gets worse (if not aligned) with scale from 1 to 64 nodes. In Section 5.4.2 we evaluate the impact of alignment (vs. random placement) in deployments of the SPS and KVS platforms ranging from 1 to 64 AWS nodes. In Section 5.4.3 we focus on the dynamic alignment between SPS and KVS deployed on the same nodes and demonstrate the speed at which performance benefits are realized over time. In Section 5.4.4 we examine a scenario where SPS and KVS are initially deployed on different nodes and thus Amoeba needs to trigger data migration to co-locate them prior to aligning partitioning schemes. Finally, in Section 5.4.5 we evaluate Amoeba's support for coordinating elasticity actions: From an originally co-located and aligned setup, the SPS triggers an elasticity action. To restore an optimized configuration, Amoeba will trigger a matching KVS elasticity action, followed by partition alignment.

We carefully instrument each operator of the application graph (Figure 5.4) to measure the throughput (processed tuples per second) and operators processing latency. The latter metric refers to the duration from the time a tuple enters the operators' processing task until the corresponding output is produced. For operators accessing external state, processing latency includes the cost of (local or network) data access to the KVS. Our evaluation

4-nodes	8-nodes	16-nodes	32-nodes	64-nodes
24.36%	12.82%	6.04%	3.09%	1.71%

Table 5.3: Local-hit rate per node for different cluster sizes

focuses on the Toll operator, as it is the most heavily-loaded task accessing the external database, and on the critical path of the application. We thus consider it representative of overall application performance.

All reported results are the average of at least 5 runs of each experiment with a standard deviation of <1%.

5.4.1 Unaligned access gets worse with scale

In this section we study the data access pattern from the Flink processing task to Redis data store partitions. In contrast to the previous sections where we reported the aggregate throughput for all Toll operator instances, we now focus on each individual node of the cluster. Each Toll operator instance, hosted on a different node, issues data requests to Redis. Each request can be served by the local (co-located with the Flink task) Redis instance or by a remote instance hosted on another node in the cluster. We define the *local-hit rate* metric as the percentage of the total data-requests issued by an operator instance towards Redis that are served by the local Redis instance. Without proper partition alignment through Amoeba, partitions are randomly placed across cluster nodes. While each system may individually achieve good load balance, lack of coordination leads to high remote access penalizing overall performance. In this section we study the distribution of data requests of a single node across all Redis instances and its performance impact.

Both systems use a hash-based approach in their partitioning scheme which is very popular in many modern data processing systems. Flink's method is based on MurmurHash algorithm while Redis uses a CRC-32 based algorithm. Table 5.3 shows the average local-hit rate per node for different cluster sizes. The local-hit rate closely follows the 1/N probability distribution with N the number of Redis instances (equivalent to the cluster size).

Figure 5.5 depicts the correlation of the *per node* local-hit rate and per node through-

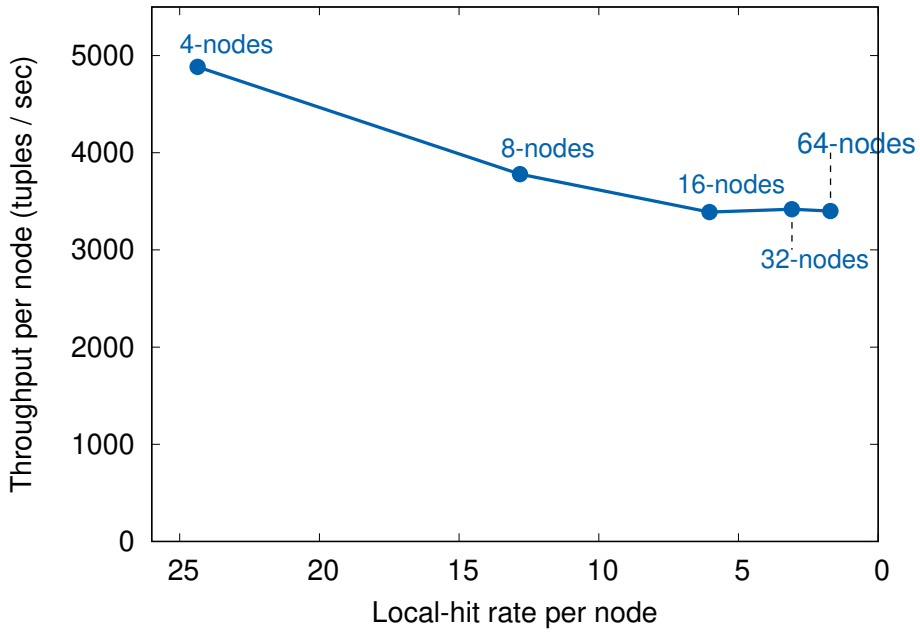


Figure 5.5: Clusters with more nodes result to lower local-hit rate affecting the throughput per node. Clusters with more than 16 nodes are close to the worst case where almost all requests are served by remote KVS instances

put for different cluster sizes. While the aggregate throughput increases as the system scales (Figure 5.7), the data requests of each processing task that happen to be served by the local Redis instance decrease in inverse proportion to the cluster size, penalizing the performance of each node. As more Redis instances join the cluster, the probability of hitting the local instance decreases even more. However, scaling beyond 16 nodes does not seem to reduce per node throughput further as the local-hit rate is already close to the worst-case scenario where nearly all requests are served by remote Redis instances.

In contrast, aligning systems through Amoeba results in 100% local hit rate, leading to high performance of each node.

5.4.2 Benefits of alignment

To evaluate the performance overhead of remote data access we first deploy the application with parallelism one, where each operator is deployed as a single instance (single

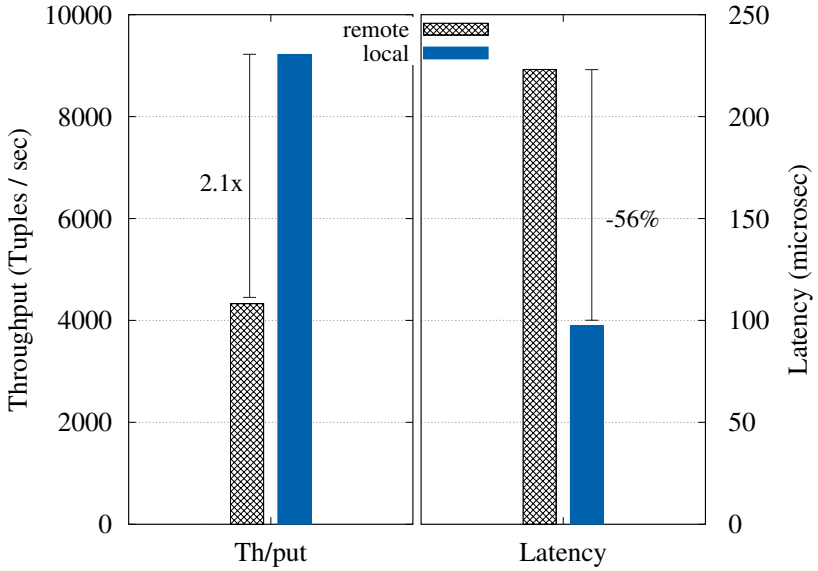


Figure 5.6: Single-partition deployment. Improving data locality results to 2.1x higher throughput and 2.2x lower processing time on Toll operator

partition). Redis is also deployed in standalone mode (single node), either on the same node as Flink (*local*) or on a different node (*remote*). Figure 5.6 depicts the throughput of the Toll operator in the local vs. remote cases. Co-location leads to a 2.1x speedup in throughput. Figure 5.6 shows also the latency (processing time) of the Toll operator, which includes the computation as well as the cost of data access. Co-location leads to 56% lower latency compared to remote data access, avoiding the cost of a network round trip. The base network latency is measured at 11.25μs using qperf [6].

Next we examine the impact of local vs. remote access to Redis tables in scaled deployments of 4 to 64 AWS nodes. Each processing operator is partitioned to multiple instances and each instance is assigned a non-overlapping key range. Each node hosts a Flink task manager and a Redis instance, with each Flink operator instance and Redis partition instance hosted on a separate node. By default each system follows its own partitioning scheme and partition assignment method across nodes in the cluster. This leads to cross-talk communication over the network as some of the data requests are served by the local (co-located) Redis instance while others are served by remote Redis instances (the proba-

[6]https://github.com/linux-rdma/qperf (retrieved September 2021)

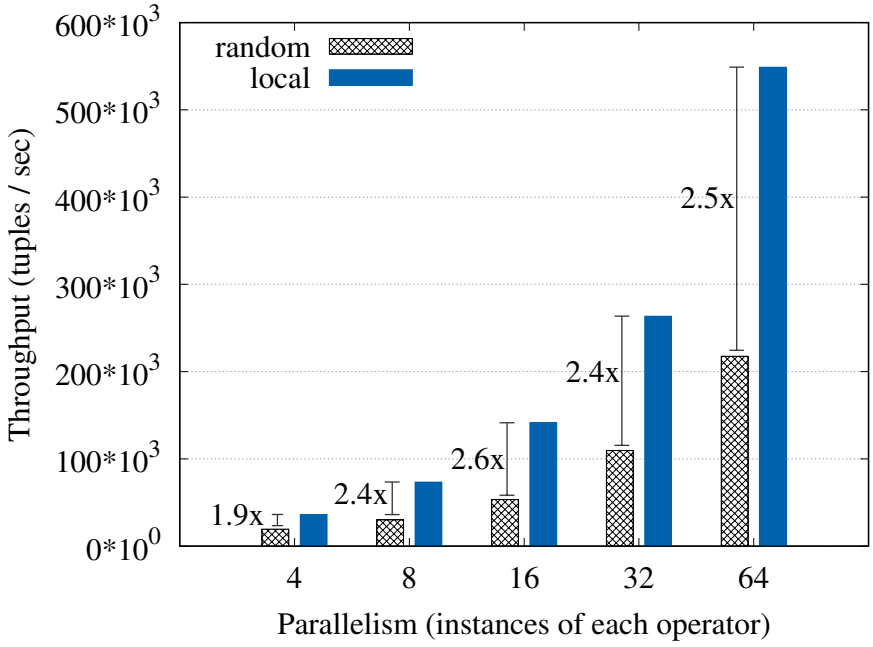


Figure 5.7: Throughput of Toll (map). Amoeba improves overall throughput consistently for cluster sizes 4-64 AWS nodes

bility of local access is inversely proportional to the number of partitions). Figure 5.7 depicts the aggregated throughput of all Toll operator instances as we scale our deployment from 4 to 64 nodes. The figure compares the unaligned case (*random*) where each system uses its own partitioning scheme and partition placement across nodes, to the system resulting from Amoeba partition-alignment (*local*), re-assigning Flink processing partitions to operator instances across nodes. We observe that Amoeba coordination more than doubles the achieved throughput in all cases, reaching up to 2.6x improvement. This experiment provides evidence that partition-alignment benefits persist with increasing cluster sizes.

5.4.3 Dynamic alignment of partition schemes

In this section we demonstrate the ability of Amoeba to dynamically adapt SPS-KVS key routing based on the partitioning schemes it dynamically receives by the Coordinator. The

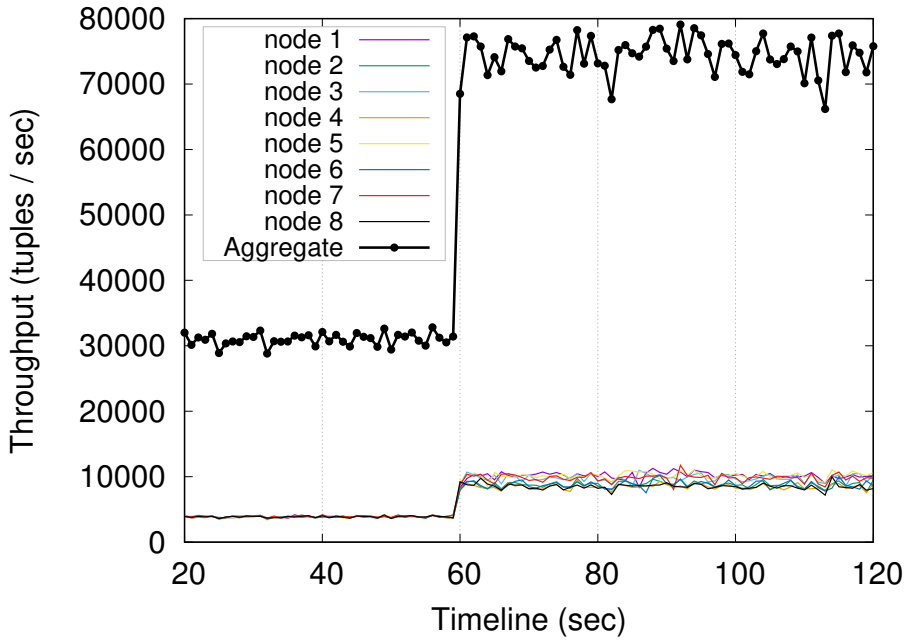


Figure 5.8: Dynamic adaptation: our adaptive SPS partitioner contacts the Amoeba coordinator and applies its alignment plan improving the throughput during the run

Coordinator periodically probes the Flink Job Manager and Redis instances in the cluster to discover the partitions of each system and their distribution within the cluster. Amoeba applies its aligning actions whenever it discovers opportunities to improve data locality. For the sake of this demonstration we configure our adaptive partitioner to contact the Coordinator for the first time 60 seconds into a run (to demonstrate the impact of adaptation actions during the run) with a Flink-Redis deployment over 8 AWS nodes. Figure 5.8 presents the performance improvement achieved as the adaptive partitioner effects the optimal alignment between systems at run time. Sixty seconds into the run we see a 2.4x improvement on the aggregated throughput as well as on the throughput per node in the 8-node cluster. Note that adaptation is seamless, exhibiting no downtime.

Eliminating the cost of data access over the networks, results in higher processing rate of its input queue and routing its output to the downstream operator. Figure 5.9 depicts the amount of data transferred in and out of a single node of the cluster and the CPU

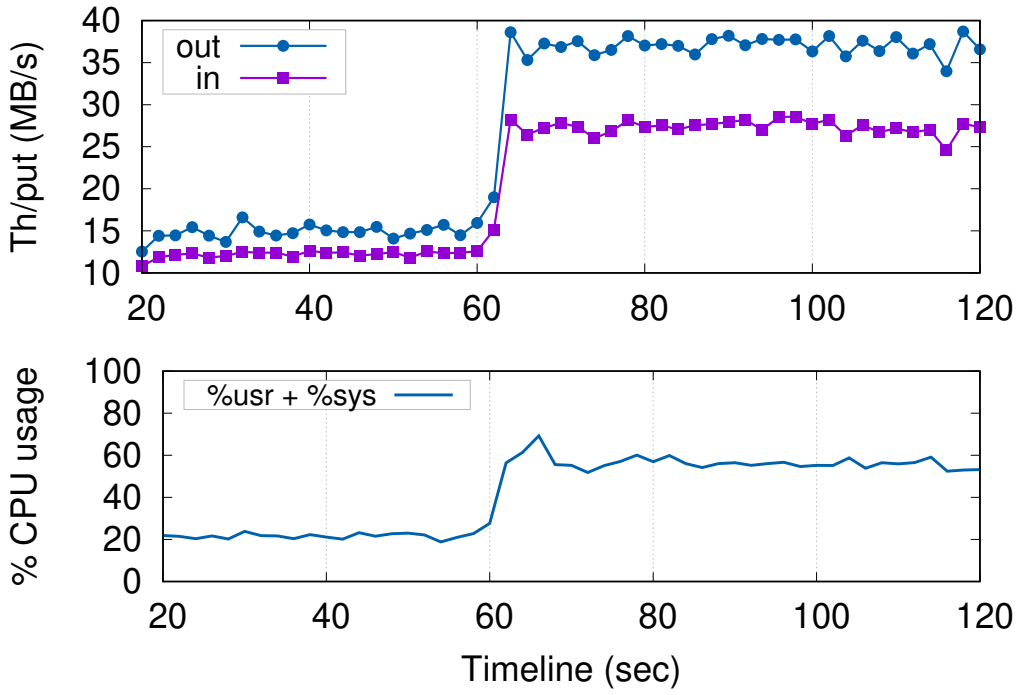


Figure 5.9: Network traffic and CPU utilization of a cluster node. Alignment improves resource utilization

utilization (user and system level) before and after the alignment of the systems. As the node is able to improve its processing capacity, it exhibits higher resource utilization. All nodes in the cluster show similar resource usage trends to that depicted in Fig. 5.9.

5.4.4 Combining alignment with data migration

The Amoeba coordinator takes into account topology (placement) metadata of both systems and possible constraints and generates an alignment plant to improve overall performance (Section 5.2). When systems are co-located in the same set of nodes or if the system is allowed to move the processing tasks on the nodes hosting the KVS, the adaptation is seamless as it usually requires little data migration across nodes. However, this is not always the case. Here we demonstrate a scenario where the SPS operators and the KVS partitions are deployed on a disjoint set of nodes, and the nodes hosting the data

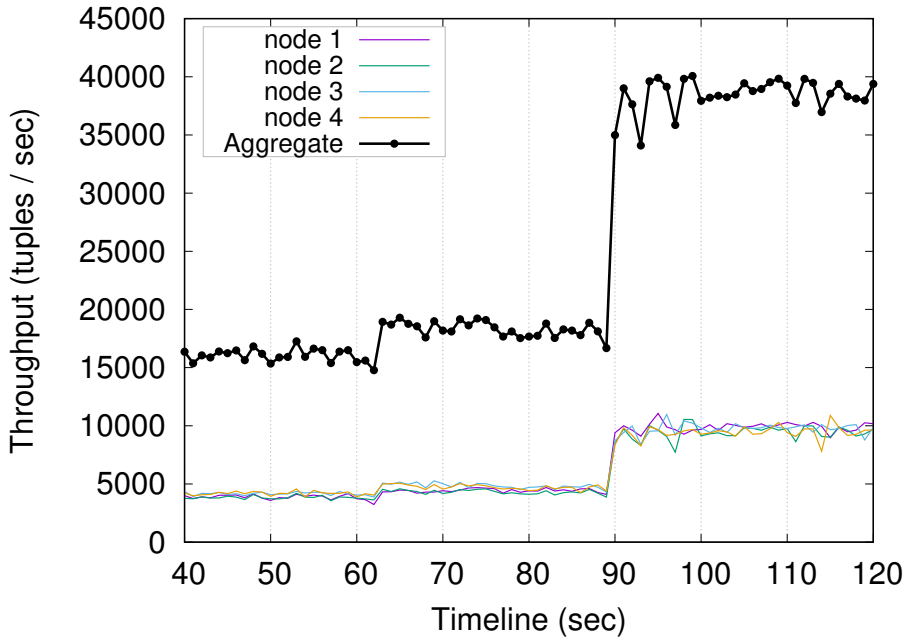


Figure 5.10: An alignment plan that includes data migration. Amoeba first migrates data closer to the processing tasks, then aligns the partitioning scheme without any downtime

(KVS) are dedicated storage nodes, thus not a target for placing or migrating processing tasks. Therefore in this case the only option to achieve alignment (locality) is to move KVS data partitions onto the nodes hosting stream-processing tasks. This experiment focuses on the 4 nodes that are initially hosting only SPS partitions.

Our Redis driver allows data migration to the SPS nodes without any noticeable service disruption by leveraging Redis data-replication mechanisms (available in many other KVSs). To realize an alignment plan that migrates KVS data close to SPS operators, Amoeba creates new Redis follower replicas (termed slaves in Redis) on nodes hosting SPS tasks, and performs state transfer to bring these new replicas up to date. As soon as followers catch up with primaries, Amoeba triggers a primary-switch [152] promoting followers to primaries. Having both processing tasks and data shards co-located on the same set of nodes, the next step of the adaptation plan is to align the SPS partitioning scheme to achieve locality in data access, as demonstrated in Section 5.4.2.

Figure 5.10 depicts the dynamic adaptation of the system in this scenario. To clearly demonstrate the impact of each alignment step (data migration, then partition-scheme alignment) we configured the Amoeba coordinator to start the alignment process 60 seconds into the run and space alignment actions 30-seconds apart. Starting the experiment using a different set of nodes for processing operators and KVS partitions (interval 0-60 sec), requests are served by remote Redis instances incurring a performance cost for data accesses to external state over the network. At 60 seconds, the Amoeba coordinator creates the alignment plan based on the topology metadata and instructs the Redis driver to create follower replicas on the SPS nodes according to the plan.

In the interval 60-90 seconds, follower nodes catch up with their primaries, and are then promoted to primaries and start serving requests. There is no service disruption during this period, in fact we observe a 23% higher throughput as some of the data requests are served by a co-located Redis instance (in Section 5.4.1 we saw that local-hit rate is non-zero even without partition alignment in the systems). However the performance can be further improved: The final step of the plan is realized 90 seconds into the run, when our Flink partitioner adapts its partitioning scheme according to the plan, leading to 100% local-hit rate improving overall performance by 2x compared to the co-located but unaligned Redis instances (consistent with the results reported in Figure 5.7) or 2.38x compared to the case of remote Redis nodes.

5.4.5 Coordinating elasticity actions

Amoeba continuously strives to achieve alignment of elasticity actions between SPS and KVS to preserve locality achieved in previous configurations. Here we demonstrate a scenario where Amoeba notices a change in a fully aligned system that is stired by an elasticity action decided and effected by the SPS. Amoeba will then respond by triggering a corresponding action for the KVS and then re-adjust the SPS partitioning scheme to restore data locality.

The initial configuration includes 3 nodes, each hosting a Flink task manager and a Redis instance. For the sake of demonstration we intentionally space adaptation actions

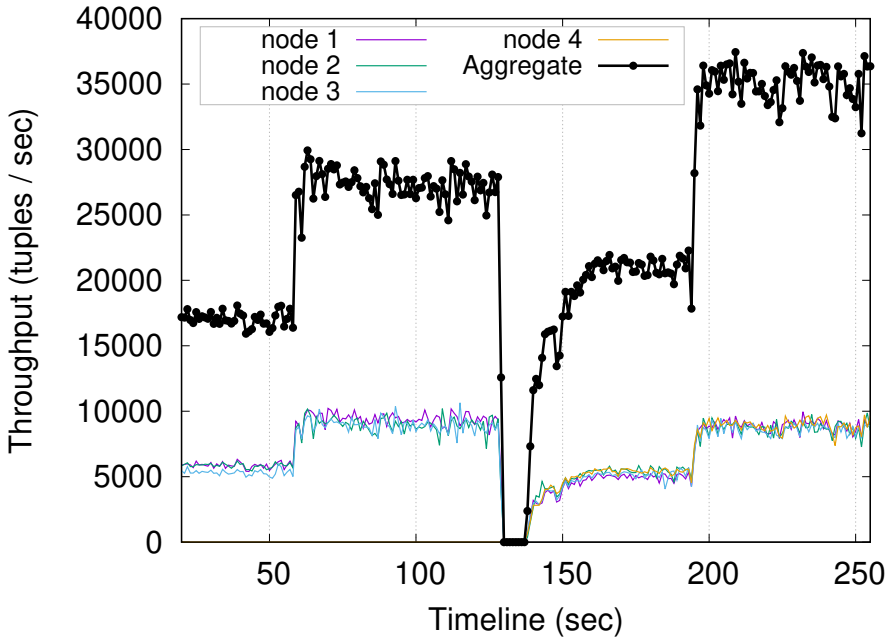


Figure 5.11: Amoeba matches elasticity actions of the SPS with corresponding actions in the KVS to maintain alignment

60 seconds apart. Figure 5.11 depicts the different stages of a run and the impact of alignment steps triggered by Amoeba. The experiment starts with 3 nodes (0-60 sec), with no coordination between systems, with each system using its own partitioning scheme and placement policy, achieving a throughput of about 17,000 tuples/sec. At 60 seconds the Amoeba coordinator aligns the key-partitioning scheme of the SPS to that of the KVS, improving overall performance in the 3 node-cluster to 27,000 tuples/sec (60-120 sec).

Next at about 130 seconds, an elasticity controller[7] of the SPS (external to Amoeba) triggers a reconfiguration action, scaling the SPS to 4 cluster nodes (Figure 5.11). This elasticity action is effected on the Flink streaming job by externally triggering a *savepoint* (a consistent image of the execution state of a streaming job [15]), then stopping and restarting the job from that savepoint with a new parallelism property. During its elasticity action (130-137 seconds in Figure 5.11), the SPS stops processing tuples. Right after the job resumes processing, the system achieves about 21,000 tuples/sec. Amoeba automatically

[7]The nature of such a controller is beyond the scope of our work

discovers that the new Flink configuration disrupts the locality achieved by previous actions. To re-align the SPS and KVS it first triggers a reconfiguration action on the KVS, creating a new Redis instance on the new node, and then triggers a background rebalancing action on the Redis cluster. The rebalancing action does not result in measurable downtime as Redis can serve requests while there are active data migrations. When the rebalancing action is over, the system reaches a stable state where both systems have a processing operator and a KVS instance on each node. In the last step of the alignment plan, Amoeba instructs the SPS adaptive partitioner to apply the new partitioning scheme improving the overall system performance (at around 190 seconds), reaching about 35,000 tuples/sec. Note that had Amoeba actions not been spaced for demonstration, the throughput increase would have happened much more rapidly.

Figure 5.11 also depicts that without coordination, the *per node* throughput is affected by the cluster size, i.e. the *per node* throughput for the 4-node cluster (135-190 seconds) is lower than the 3-node cluster (0-60 seconds) (as analyzed in Section 5.4.1). Amoeba discovers changes in the topology and produces alignment plans to preserve locality, resulting in high throughput and stable per-node performance.

As Amoeba's actions overlap in time with actions of external management systems, such as the elasticity controller in this experiment, a controller should be aware (via a distributed policy coordination solution [77]) that Amoeba adaptation actions are in progress and factor-in their delay when probing the effect of its actions in its control loop.

5.5 Summary

In this chapter we presented a broad investigation of the benefits of Amoeba, a system we designed and implemented for exploiting data locality between processing tasks (SPS) and external state (KVS) via dynamic adaptivity actions (appropriate data or task placement, data or task migrations, alignment of partitioning schemes, and coordination of elasticity actions). Our experimental evaluation of Amoeba coordinating various deployments of Flink and Redis executing the Linear Road application, demonstrate significant (up to 260%) performance improvements in large-scale setups of up to 64 AWS nodes by re-

ducing cross-talk between processing tasks and the corresponding external state over the network. To the best of our knowledge, dynamic adaptation to achieve collocation in this context has not been explored in previous work. Control over data placement may at times require adaptation actions requiring state redistribution, as in elasticity actions involving stateful operators. In our ongoing work we are pursuing the integration and coordination of efficient elasticity mechanisms for KVSs and SPSs [87, 112, 151, 191] in SPS-KVS systems managed by Amoeba. We are also exploring the integration of Amoeba into elasticity controllers to achieve concerted actions. Where integration is not feasible or impractical, we will explore techniques proposed in the context of autonomic computing [77] for the distributed coordination of resource management policies.

Chapter 6

Impact of technology improvements on the proposed methods

Ever since mankind invented computing devices, technology evolves and computing systems get ever more advanced. Advances in computer hardware since at least the 1960s have enabled the evolution of data management from paper-based manual processing to modern data management systems. This progress in the capabilities of hardware resources is expected to continue in the future. Technology advances are expected to improve the performance of data stores and aid their capacity to support big-data processing, via more storage capacity, faster access to persistent storage devices, higher processing power for storing and retrieving data, and faster and more efficient network data transfers [59]. In this chapter we discuss how technology evolution and trends may affect the challenges we focus on in this dissertation and impact the adaptation mechanisms we study.

6.1 Incremental elasticity

The data store expansion actions we studied in Chapter 2 involve network data movements across nodes. In preparation of such network transfers, we often observe significant I/O activity to prepare the data to be shipped. The evolution of storage technology espe-

cially on the storage devices (such as Flash-based disks and I/O interfaces (e.g. NVMe [30]), which will allow high I/O throughput and lower data access latency data transfers, is expected to reduce I/O overhead in elasticity actions. High-speed network communication (such as InfiniBand [19]) will also allow for faster and lower-overhead data transfers. Networking with minimal processing requirements (such as RDMA [34]) would reduce the processing cost of data transfers on both senders and receivers. In addition, we expect significant growth in the per-node processing capacity, even on storage-optimized nodes. While technology advances are expected to improve data-transfer speeds and to reduce overheads, there are two factors that support the projection that the conclusions of the thesis are expected to persist in the future: First, the amount of data maintained by the data stores is expected to be ever-increasing, keeping pace with the expected growth in storage, processing, and networking capacity. Second, one of the key observations and benefits in the incremental elasticity technique, namely the ordering of data transfers such that the joining nodes benefit the system as early as possible is still expected to be valid. The combination of lower-overhead data transfers and more processing capacity available at joining nodes strengthens the promise of incremental elasticity, providing additional benefits from early serving of data while network transfers are still in progress.

6.2 Replica-group reconfiguration

In Chapter 3 we studied the impact of internal data reorganization and garbage collection tasks, as well as external background activities such as data backup actions, on the performance of replicated data stores. Such background tasks pose high CPU and I/O overheads on the data stores. Projected improvements in the processing power and/or on the performance and overhead of the I/O path is expected to be matched by the anticipated growth in the amount of data stored, maintaining a sizeable performance impact due to such activities. Thus we believe that the proposed replica-group reconfiguration technique would not be significantly impacted by technology advances in terms of processing, I/O and networking, and remain a key adaptation technique to mask the performance overhead of background activities.

In Chapter 3 this dissertation further contributes a mechanism for maintaining fresh read caches in non-read-serving replicas, and demonstrates that the performance benefit indeed depends on storage technology (SSD vs. hard disk) as measured in recovery time (time to recover to the pre-reconfiguration performance, Section 4.3). Disruptive technology advances that could bridge the performance gap between volatile memory and persistent storage (such as Intel Optane [20]) promise to change the landscape in the memory-storage hierarchy, significantly reducing the memory-storage performance gap. In this case, the impact of the mechanism for maintaining fresh read caches in non-read-serving replicas that was proposed in Chapter 3 may be reduced.

6.3 Improving data locality when accessing external state

In Chapter 5 of this dissertation we built upon the idea that co-location of the processing tasks and data-store partitions can reduce the latency of accessing external state and thus improve overall performance. We address the need to achieve such co-location and to maintain it through independent changes within inter-dependent distributed middleware systems via data migration, data replication, and data partitioning mechanisms. The benefits achieved through these contributions fundamentally rely on the performance gap between local and remote (network) communication, assuming data are already in memory (as is typically the case when using in-memory data stores, such as Redis). Projected improvements in high speed, low latency networking may reduce the cost of remote data access, especially if one takes into account a possibly slower growth in local-memory access speeds. However, baring disruptive technologies that could introduce drastic changes to network communication (especially in the host and network interface components, as these typically dominate latency in high-performance networks), we expect that the local vs. remote network communication performance difference is expected to persist in the future, sustaining the impact of the contributions described in Chapter 5.

Chapter 7

Conclusions

Large-scale distributed data stores are essential building blocks of modern application services, persisting and managing application state. Such systems have evolved considerably in recent years along several directions, leading to a number of different data stores proposed and available today. This dissertation focuses on the ability of such systems to adapt so as to keep up with internal or external changes they face, posing different performance challenges over their lifecycle. In this dissertation we specifically study a range of challenges a data store faces and propose novel adaptation mechanisms and improvements to existing such mechanisms to avoid performance impact and sustain service-level objectives. The dissertation addresses performance challenges over data-store adaptation actions during internal or external changes, which had not been addressed so far. The contributions of this work are primarily driven by the need to sustain or improve system performance through optimizations that mask overheads, allowing systems to quickly and smoothly adapt to changes. The mechanisms contributed in this thesis are orthogonal to and do not affect other system aspects such as data availability, durability or consistency.

Distributed data stores face an additional burden over stateless distributed systems during changes in system configuration, in that they have to rebalance large amounts of durable data by migrating them across nodes during elasticity actions. In this dissertation we study the efficiency of such elasticity actions and challenges in rapidly and efficiently expanding capacity, which had not been addressed so far. In Chapter 2 we identified per-

formance overheads that can hurt overall performance during data movements. Thus, the store often fails to achieve its performance-oriented goals while transitioning to the new configuration. This dissertation advances the state of the art closer towards truly elastic distributed data stores. In Chapter 2 we proposed incremental elasticity, a new mechanism that orchestrates data movements ensuring that there are fewer nodes involved in network transfer at any time, reducing the overall performance drop during elasticity. As there is a trade-off between the total duration of the elasticity action and its impact, our mechanism ensures that the system progressively benefits by increasing its capacity during the reconfiguration. We perform fine-grain data movements ensuring that as soon as a transfer is over, the associated data are becoming available for access on the new node, while a subsequent transfer of data takes place. Overall, the system can effectively get capacity improvements earlier while the performance impact is lower. This is complementary to solutions that study the provisioning and capacity planning of the system.

This dissertation also contributes to more stable data store performance. In Chapter 3 we studied different internal and external sources of performance overheads that result in performance variability. As these actions cannot always be avoided or postponed for too long, they cause significant I/O activity and corresponding CPU overhead. When they happen and depending on their severity, a system may occasionally fail to sustain the expected service-level performance objectives. In Chapter 3 we proposed a lightweight adaptation action that allows replica-group members to dynamically change roles (i.e. primary replica, secondary replica) to hide the performance bottlenecks imposed by data store's internal and external background tasks, thus reducing performance variability. Even though other solutions, such as single-node data structure optimizations, may offer alternative ways to approach this problem, this dissertation explored replica-group reconfiguration as complementary to solutions that may offer single-node performance improvements.

Replica-group reconfiguration actions occur for many reasons such as performance enhancements (as we described in Chapter 3) but also due to failures, load balancing and workload migration. In the course of studying replica-group reconfiguration actions, we observed a performance glitch that, although small relative to the impact of the challenges addressed in Chapter 3, nevertheless could have a lasting performance hit in certain cases.

Starting from the observation that under certain circumstances practical replicated systems restrict reads to a few replicas (typically one) to optimize for read-dominated workloads, secondary replicas cannot fetch up-to-date contents into their own cache. During reconfiguration, unprepared replicas suffer from a high cold-cache effect leading to high latency spikes and throughput drops for significant amounts of time (several minutes). Chapter 4 studied in depth the performance impact of this observation in replicated data stores. The proposed solution results in smoother replica-group reorganization actions, allowing the system to seamlessly transition to a new configuration.

In the course of this dissertation we also observed that an inefficient data-access path between the application and data-persistence (storage system) tier in multi-tier architectures, forcing expensive network communication between them, can critically affect overall system performance. This dissertation contributes novel adaptation processes to achieving efficient cross-system communication by continuously aligning data stores with distributed middleware platforms. In Chapter 5 we built upon the idea that the performance of accessing data can be improved via co-location of the processing tasks and data partitions. In this dissertation we proposed a system that pursues opportunities for data locality as a means to reducing communication overheads by discovering topology metadata across systems and adapt data distribution across nodes (either via initial data placements or through data migrations) and aligning data partitioning schemes across systems. Coordinating data stores with application middleware through dynamic partition alignment reduces communication overheads allowing for a more efficient data-access path.

The work in this dissertation brings us closer to truly elastic data stores with lower performance impact during system reconfiguration, while the overall data store capacity progressively increases. Replicated data stores can mask performance overheads imposed by internal or external background activities that are necessary for continuous system operation over time, by dynamically changing the roles of replica-group members. This dissertation also improved the performance of replica-group reconfiguration actions themselves, which under certain circumstances can result in high performance impact during the transition to the new configuration, as some replicas engaged in the transition are impacted by cold-cache effects. The results of this dissertation allow the system to transition

to a new configuration without noticeable impact. Finally, our work contributes to better alignment in the cross-system communication between application middleware and underlying data store, systems with separate lifecycle and management policies, continuously optimizing the data access path between them.

Overall, the research contributions of this dissertation advance the state of the art in distributed data stores in the direction of systems that adapt more efficiently and in new ways through internal and external changes in their lifecycle, making an important step towards the long-term vision of autonomic systems.

Chapter 8

Directions for Future Work

Adaptation in large scale distributed data stores covers a vast research space. The focus of this dissertation was primarily driven by the need to sustain or improve system performance during specific internal or external changes in different phases of the lifecycle of a data store. Due to the breadth of the research space, there are still several challenges to be addressed. In this chapter we point to several avenues for further research work.

A basic form of adaptation is via tuning of the configuration settings or customization knobs offered by a data store. System administrators rely on ”best practice” heuristics, statistics or trace analysis to improve the system performance. However, this is a tedious and time consuming process. While there has been significant work on advisors that can tune the system based on the workload and data characteristics [45, 73] or even choosing indexes [74,105,185] or partitioning schemes, these settings are usually fixed and do not allow dynamic adaptation as the system evolves throughout its lifecycle. Methods for online adaptation of system configuration is a challenging research area.

In addition, the adaptation options are limited by the set of knobs exposed by each specific system. Automatic and deep exploration of systems to discover system properties that allow the customization can bring us closer to autonomous systems. We believe that adaptation in data stores can go further allowing for adaptation of core components, such as the internal data representations. Different data structure designs offer efficient data access paths for specific workload characteristics [56, 113]. Modern state-of-the-art data

stores are designed around a fixed data structure. However, as data stores face diverse workloads with different access patterns and requirements, they need to be able to adapt the underlying data structures. Data stores that are flexible to accommodate multiple data structures and dynamically adapt the storage layout and the access patterns can improve its performance during its lifespan while the workload access pattern evolves. Materializing data into different layouts on-the-fly is a challenging task [48,55]. This is an interesting research area that can lead to results that are complementary to those contributed by this thesis.

In addition, specific data structures make core design choices to improve the efficiency of data access. These choices should be driven by the access pattern the data store faces. State-of-the-art data stores, however, build around generic and static options. Recent work, limited to certain aspects of specific data structures (LSM-tree), has shown that the system can improve its performance and resource usage by balancing between the cost of updates, lookups and the storage space [83,84].

Data stores could go even further and automatically generate a data structure design that best fits the workload requirements [114]. This goes far beyond configuring a specific system, as the combination of design primitives of data structures [113] could lead to custom data structures and algorithms previously unknown that could improve the systems performance under the custom workload characteristics it serves.

Data stores can incorporate machine learning techniques to learn and discover opportunities for adaptation. Replacing the traditional control-flow computation with data-dependency-focused computations will allow machine-learning methods to empower systems. Learning-based models will have more flexibility to explore trade-off and discover tuning opportunities for significant performance gains [128,129]. Data stores can leverage the wide adoption of specialized hardware optimized for machine learning.

The research space revolving around adaptivity aspects of distributed data stores has offered us many exciting endeavors and is not short of more challenges, we thus encourage further work in this space. We are looking forward to the discovery of new adaptation mechanisms, hopefully building upon and extending the research results of this thesis, that allow for autonomic data management systems that guarantee system performance

despite the large (and unavoidable) degree of variability in our evolving Internet-driven world.

Bibliography

[1] Akamai Online Retail Performance Report: Milliseconds Are Critical. https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp. Accessed: 06/2021. (Cited in Section 1.1.)

[2] Amazon DynamoDB. https://aws.amazon.com/dynamodb/. Accessed: 06/2021. (Cited in Sections 1.3.1 and 2.)

[3] Amazon Found Every 100ms of Latency Cost them 1% in Sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/. Accessed: 06/2021. (Cited in Section 1.1.)

[4] Amoeba (wikipedia). https://en.wikipedia.org/wiki/Amoeba. (Accessed 06/2021). (Cited in Section 1.)

[5] Basho Riak NoSQL database. https://riak.com/products/riak-kv/. Accessed: 11/2019. (Cited in Sections 1.1, 2, 2.1, and 2.4.)

[6] Bing and Google Agree: Slow Pages Lose Users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html. Accessed: 06/2021. (Cited in Section 1.1.)

[7] Blink: How alibaba uses apache flink. https://www.ververica.com/blog/blink-flink-alibaba-search. (Accessed 06/2021). (Cited in Section 5.)

[8] Bootstrapping performance improvements for Leveled Compaction. http://www.datastax.com/dev/blog/bootstrapping-performance-improvements-for-leveled-compaction. Accessed: 06/2021. (Cited in Sections 2 and 2.3.1.)

[9] Cassandra Basics. https://cassandra.apache.org/_/cassandra-basics.html. Accessed: 09/2021. (Cited in Section 2.)

[10] Facebook RocksDB. http://rocksdb.org. (Accessed: 06/2021). (Cited in Sections 1.3.2, 2, 1.4, 3, 3.1, 3.1, and 3.2.)

[11] Falling in and out of love with DynamoDB. https://blog.0x74696d.com/posts/falling-in-and-out-of-love-with-dynamodb-part-ii/. Accessed: 06/2021. (Cited in Sections 1.3 and 2.)

[12] Flink event time. https://ci.apache.org/projects/flink/flink-docs-stable/dev/event_time.html. (Accessed 06/2021). (Cited in Section 5.3.1.)

[13] Flink operators. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators. (Accessed Mar 2021). (Cited in Section 5.2.3.)

[14] Flink queryable state (beta). https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/queryable_state.html. (Accessed 06/2021). (Cited in Section 5.1.2.)

[15] Flink savepoints. https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/savepoints.html. (Accessed 06/2021). (Cited in Sections 5.2.3 and 5.4.5.)

[16] Google LevelDB. https://github.com/google/leveldb. (Accessed: 06/2021). (Cited in Sections 3 and 3.1.)

[17] How are read requests accomplished. https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlClientRequestsRead.html. Accessed: 09/2021. (Cited in Section 2.3.1.)

[18] How data is distributed across a cluster (using virtual nodes). "https://docs.datastax.com/en/cassandra/3.x/cassandra/architecture/archDataDistributeDistribute.html". Accessed: 06/2021. (Cited in Section 2.1.)

[19] InfiniBand Trade Association. https://www.infinibandta.org. Accessed: 09/2021. (Cited in Section 6.1.)

[20] Intel® Optane™ Memory - Responsive Memory, Accelerated Performance. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html. Accessed: 09/2021. (Cited in Section 6.2.)

[21] Keystone real-time stream processing platform at netflix. https://netflixtechblog.com/keystone-real-time-stream-processing-platform-a3ee651812a. (Accessed 06/2021). (Cited in Sections 1.3.4 and 5.)

[22] Leveled compaction. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction. (Accessed: 06/2021). (Cited in Sections 1.3.2, 3.1, and 3.3.1.)

[23] Mongodb. https://www.mongodb.com. (Accessed 06/2021). (Cited in Sections 1.1, 2, 1.4, 3.1, and 3.2.)

[24] MongoDB Checkpointing Issues. https://goo.gl/idyog2. (Accessed: 06/2021). (Cited in Section 3.)

[25] Mongodb server manual: Read preference. https://docs.mongodb.com/manual/core/read-preference/. accessed 06/2021. (Cited in Section 4.3.)

[26] MongoDB Wiki: replication internals. https://github.com/mongodb/mongo/blob/master/src/mongo/db/repl/README.md. (Accessed: 06/2021). (Cited in Sections 3.1 and 3.2.)

[27] Mongodump. https://docs.mongodb.com/manual/reference/program/mongodump/. (Accessed: 06/2021). (Cited in Section 3.3.2.)

[28] Mongorocls. https://github.com/mongodb-partners/mongo-rocks. (Accessed: 06/2021). (Cited in Section 3.2.)

[29] NoSQL, NewSQL and Beyond: The drivers and use-cases for database alternatives. https://451research.com. Accessed: 06/2021. (Cited in Sections (document) and 1.1.)

[30] NVM Express. https://nvmexpress.org. Accessed: 09/2021. (Cited in Section 6.1.)

[31] Read repair. "https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html". Accessed: 11/2019. (Cited in Section 2.1.)

[32] Redis. https://redis.io. (Accessed 06/2019). (Cited in Sections 1.1 and 1.4.)

[33] Redis cluster specification. https://redis.io/topics/cluster-spec. (Accessed Mar 2021). (Cited in Section 5.2.3.)

[34] Remote Direct Memory Access (RDMA) over IP Problem Statement. https://datatracker.ietf.org/doc/rfc4297/. Accessed: 09/2021. (Cited in Section 6.1.)

[35] Riak a successful failure. https://www.slideshare.net/GiltTech/riak-a-successful-failure-11512791. Accessed: 06/2021. (Cited in Section 2.4.)

[36] Scylladb. https://www.scylladb.com. (Accessed: 06/2021). (Cited in Section 3.)

[37] The Value of a Millisecond: Finding the Optimal Speed of a Trading Infrastructure. https://research.tabbgroup.com/report/v06-007-value-millisecond-finding-optimal-speed-trading-infrastructure. Accessed: 06/2021. (Cited in Section 1.1.)

[38] TPC-C. http://www.tpc.org/tpcc/. accessed 06/2021. (Cited in Sections 4.3 and 4.3.11.)

[39] TPC-H. http://www.tpc.org/tpch/. accessed 06/2021. (Cited in Sections 4.3 and 4.3.11.)

[40] Using DynamoDB in production. https://www.dyspatch.io/blog/using-dynamodb-production/. Accessed: 06/2021. (Cited in Sections 1.3 and 2.)

[41] Why does speed matter? https://web.dev/why-speed-matters/. Accessed: 06/2021. (Cited in Section 1.1.)

[42] Dynamic snitching in cassandra: past, present, and future. https://www.datastax.com/blog/2012/08/dynamic-snitching-cassandra-past-present-and-future, accessed 06/2021. (Cited in Section 4.1.)

[43] How are read requests accomplished? https://docs.datastax.com/en/ddac/doc/datastax_enterprise/dbInternals/dbIntClientRequestsRead.html, accessed 06/2021. (Cited in Section 4.1.)

[44] Riak. load balancing and proxy configuration. https://docs.riak.com/riak/kv/2.2.3/configuring/load-balancing-proxy, accessed 06/2021. (Cited in Section 4.1.)

[45] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 359–370, New York, NY, USA, 2004. Association for Computing Machinery. (Cited in Section 8.)

[46] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013. (Cited in Sections 5 and 5.1.2.)

[47] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric

Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015. (Cited in Sections 5 and 5.1.)

[48] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2o: A hands-free adaptive store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Snowbird, Utah, 2014. (Cited in Sections 1.5 and 8.)

[49] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. Storage management in asterixdb. *Proc. VLDB Endow.*, 7(10):841–852, June 2014. (Cited in Section 3.)

[50] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 577–588, New York, NY, USA, 2013. Association for Computing Machinery. (Cited in Sections 5 and 5.1.1.)

[51] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, USA, 2002. USENIX Association. (Cited in Section 5.2.2.)

[52] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004. (Cited in Sections 1.3.4, 4, 5, 4, 5.3, 5.3.1, 5.3.1, and 5.3.2.)

[53] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI 14), pages 367–381, Broomfield, CO, October 2014. USENIX Association. (Cited in Sections 1.1, 1.3.3, 4, 4.1, and 4.3.11.)

[54] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. (Cited in Section 1.1.)

[55] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 583–598, New York, NY, USA, 2016. Association for Computing Machinery. (Cited in Sections 1.5 and 8.)

[56] Manos Athanassoulis, Michael Kester, Lukas Maas, Radu Stoica, Stratos Idreos, Anastassia Ailamaki, and Mark Callaghan. Designing access methods: The rum conjecture. In *International Conference on Extending Database Technology (EDBT)*, Bordeaux, France, 2016. (Cited in Section 8.)

[57] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM. (Cited in Sections 1.2, 4, and 4.3.6.)

[58] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB*, 5(8):776–787, April 2012. (Cited in Section 4.)

[59] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017. (Cited in Section 6.)

[60] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3), 2018. (Cited in Section 4.1.)

[61] Nikos Batsaras, Giorgos Saloustros, Anastasios Papagiannis, Panagiota Fatourou, and Angelos Bilas. Vat: Asymptotic cost analysis for multi-level key-value stores. *arXiv arXiv:2003.00103*, 2020. (Cited in Section 1.3.2.)

[62] Samuel Benz and Fernando Pedone. Elastic paxos: A dynamic atomic multicast protocol. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2157–2164, 2017. (Cited in Sections 1.3.2 and 3.)

[63] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 141–154. USENIX Association, 2011. (Cited in Sections 1.3.2, 1.3.3, 4, and 4.1.)

[64] Aaron Brown and David A. Patterson. Towards availability benchmarks: A case study of software raid systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, page 22, USA, 2000. USENIX Association. (Cited in Section 2.4.)

[65] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Distributed systems (2nd ed.). pages 199–216. 1993. (Cited in Sections 1.3.2, 1.3.3, 3.1, 4, and 4.1.)

[66] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *SIGMETRICS P. E. Rev.*, 23(1):188–197, 1995. (Cited in Sections 4.1 and 4.2.1.)

[67] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017. (Cited in Sections 5, 5.1.1, and 5.1.2.)

[68] Maria Chalkiadaki and Kostas Magoutis. Managing service performance in the cassandra distributed storage system. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 64–71, 2013. (Cited in Sections 2.3.3 and 2.4.)

[69] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. (Cited in Sections 1.3.2, 1.3.3, 4, and 4.1.)

[70] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008. (Cited in Section 1.1.)

[71] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008. (Cited in Section 3.)

[72] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*. Springer-Verlag, Berlin, Heidelberg, 2010. (Cited in Sections 1.1 and 3.)

[73] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007. (Cited in Sections 1.5 and 8.)

[74] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. (Cited in Sections 1.5 and 8.)

[75] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1087–1098, New York, NY, USA, 2016. Association for Computing Machinery. (Cited in Sections 5 and 5.1.1.)

[76] Yong Chen, Surendra Byna, and Xian-He Sun. Data access history cache and associated data prefetching mechanisms. In *Prc. 2007 SC '07*, 2007. (Cited in Section 4.1.)

[77] D. M. Chess and J. O. Kephart. The vision of autonomic computing. *Computer*, 36(01):41–50, jan 2003. (Cited in Sections 5.2, 5.4.5, and 5.5.)

[78] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792, 2016. (Cited in Section 5.)

[79] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault-tolerant systems tolerate byzantine faults. In *Proc. of the 6th USENIX NSDI'09*, Boston, MA, April 2009. (Cited in Section 3.1.)

[80] Vinicius V Cogo, André Nogueira, João Sousa, Marcelo Pasin, Hans P Reiser, and Alysson Bessani. FITCH: Supporting Adaptive Replicated Services in the Cloud. In *IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'13)*, pages 15–28, 2013. (Cited in Section 4.1.)

[81] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery. (Cited in Sections 2.3.1, 3.3, and 4.3.)

[82] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011. (Cited in Section 1.2.)

[83] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *ACM SIGMOD International Conference on Management of Data*, 2017. (Cited in Sections 1.5 and 8.)

[84] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *ACM SIGMOD International Conference on Management of Data*, 2018. (Cited in Sections 1.5 and 8.)

[85] Biplob K. Debnath, David J. Lilja, and Mohamed F. Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*, pages 11–18, 2008. (Cited in Section 1.5.)

[86] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. (Cited in Sections 1.1, 1.1, 1.3.1, 2, 2, 2.1, 2.3.3, 2.4, 3, 4, and 4.1.)

[87] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2471–2486, New York, NY, USA, 2020. Association for Computing Machinery. (Cited in Sections 5.1.2, 5.1.3, 5.2.2, and 5.5.)

[88] Thibault Dory, Boris Mejías, PV Roy, and Nam-Luc Tran. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 37–48. Citeseer, 2011. (Cited in Section 1.2.)

[89] Thibault Dory, Boris Mejias, Peter Van Roy, and Nam-Luc Tran. Measuring elasticity for cloud databases. In *Proc. of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization*, Rome, Italy, 2011. (Cited in Section 2.4.)

[90] Jennie Duggan and Michael Stonebraker. Incremental elasticity for array databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 409–420, New York, NY, USA, 2014. Association for Computing Machinery. (Cited in Section 2.4.)

[91] Jeremy Elson and Jon Howell. Handling flash crowds from your garage. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association. (Cited in Section 1.3.1.)

[92] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012. (Cited in Section 1.2.)

[93] Y. Feng, N. Huang, and Y. Wu. Efficient and adaptive stateful replication for stream processing engines in high-availability cluster. *IEEE Transactions on Parallel and Distributed Systems*, 22(11):1788–1796, 2011. (Cited in Section 5.1.3.)

[94] Anonymized for the double-blind review process. (Cited in Section 5.)

[95] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems, 2020. (Cited in Section 5.)

[96] P. Garefalakis, P. Papadopoulos, and K. Magoutis. ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees. In *Proc. of 33rd IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, 2014. (Cited in Sections 3.1, 4.1, and 4.3.11.)

[97] Buğra Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, August 2014. (Cited in Section 5.1.3.)

[98] Lars George. *HBase: the definitive guide: random access to your planet-size data.* "O'Reilly Media, Inc.", 2011. (Cited in Section 3.)

[99] M. Ghosh, W. Wang, G. Holla, and I. Gupta. Morphus: Supporting online reconfigurations in sharded nosql systems. *IEEE Transactions on Emerging Topics in Computing*, 5(4):466–479, 2017. (Cited in Sections 5.2.2 and 5.2.3.)

[100] Mainak Ghosh, Wenting Wang, Gopalakrishna Holla, and Indranil Gupta. Morphus: Supporting online reconfigurations in sharded nosql systems. In *2015 IEEE International Conference on Autonomic Computing*, pages 1–10, 2015. (Cited in Section 2.4.)

[101] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM. (Cited in Sections 1.3.2, 1.3.3, 4, and 4.1.)

[102] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. (Cited in Section 4.1.)

[103] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP'89*. (Cited in Section 4.)

[104] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, USA, 2000. USENIX Association. (Cited in Section 2.4.)

[105] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman. Index selection for olap. In *Proceedings 13th International Conference on Data Engineering*, pages 208–219, 1997. (Cited in Section 8.)

[106] Michael Hammer. Self-adaptive automatic data base design. In *Proceedings of the June 13-16, 1977, National Computer Conference*, AFIPS '77, pages 123—-129, New York, NY, USA, 1977. Association for Computing Machinery. (Cited in Section 1.5.)

[107] Michael Hammer and Arvola Chan. Index selection in a self-adaptive data base management system. In *Proceedings of the 1976 ACM SIGMOD International Conference*

on Management of Data, SIGMOD '76, pages 1–8, New York, NY, USA, 1976. Association for Computing Machinery. (Cited in Section 1.5.)

[108] Michael Hammer and Bahram Niamir. A heuristic approach to attribute partitioning. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 93–101, New York, NY, USA, 1979. Association for Computing Machinery. (Cited in Section 1.5.)

[109] Pat Helland. Mind your state for your state of mind: The interactions between storage and applications can be complex and subtle. *Queue*, 16(3):91–121, June 2018. (Cited in Section 1.1.)

[110] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. Ready for rain? a view from spec research on the future of cloud metrics. `https://arxiv.org/abs/1604.03470`. (Cited in Section 1.3.1.)

[111] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX. (Cited in Section 1.3.1.)

[112] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.*, 12(9):1002–1015, May 2019. (Cited in Sections 5.1.2, 5.2.2, and 5.5.)

[113] Stratos Idreos, Konstantinos Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. The periodic table of data structures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 41(3):64–75, 2018. (Cited in Sections 3 and 8.)

[114] Stratos Idreos, Konstantinos Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. The data calculator: Data structure design and cost synthesis from first

principles, and learned cost models. In *ACM SIGMOD International Conference on Management of Data*, 2018. (Cited in Section 8.)

[115] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 431–442, New York, NY, USA, 2006. Association for Computing Machinery. (Cited in Sections 5, 5.3, and 5.3.2.)

[116] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS)*, 28(3):257–294, September 2003. (Cited in Section 4.)

[117] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 245–256, 2011. (Cited in Sections 3.1, 3.2, and 4.)

[118] Asya Kamsky. Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB. *Proc. VLDB Endow.*, 12(12), August 2019. (Cited in Sections 4.3 and 4.3.11.)

[119] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. Association for Computing Machinery. (Cited in Sections 2.1 and 2.4.)

[120] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proc. of 12th IEEE Int. Workshop on Quality of Service, IWQOS'04*, Montreal, Canada, 2004. (Cited in Section 2.3.3.)

[121] Flora Karniavoura and Kostas Magoutis. A measurement-based approach to performance prediction in NoSQL systems. In *Proc. of 25th IEEE Int. Symposium on the Modeling, Analysis, and Simulation of Computer and Telecom. Systems, MASCOTS'07*, Banff, Canada, 2017. (Cited in Section 2.4.)

[122] Flora Karniavoura and Kostas Magoutis. Decision-making approaches for performance qos in distributed storage systems: A survey. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1906–1919, 2019. (Cited in Section 1.2.)

[123] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. A holistic view of stream partitioning costs. *Proceedings VLDB Endowment*, 10(11):1286–1297, August 2017. (Cited in Section 5.1.3.)

[124] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003. (Cited in Section 1.2.)

[125] Nick Kolettis and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th FTCS'95*, 1995. (Cited in Section 4.1.)

[126] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. On the elasticity of nosql databases over cloud management platforms. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2385–2388, New York, NY, USA, 2011. Association for Computing Machinery. (Cited in Section 1.2.)

[127] Ioannis Konstantinou, Dimitrios Tsoumakos, Ioannis Mytilinis, and Nectarios Koziris. Dbalancer: Distributed load balancing for nosql data-stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1037–1040, New York, NY, USA, 2013. Association for Computing Machinery. (Cited in Section 2.4.)

[128] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A

learned database system. In *Conference on Innovative Data Systems Research (CIDR)*, 2019. (Cited in Section 8.)

[129] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. Association for Computing Machinery. (Cited in Section 8.)

[130] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. *Proc. of the VLDB Endowment*, 7(12):1219–1230, August 2014. (Cited in Sections 2.1 and 2.4.)

[131] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings VLDB Endowment*, 1(1):574–585, August 2008. (Cited in Section 5.1.2.)

[132] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010. (Cited in Sections 1.1, 1, 1.4, 2, 2, 2.1, 2.4, and 3.)

[133] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. (Cited in Section 4.)

[134] Butler W. Lampson. How to build a highly available system using consensus. In *Proc. of WDAG '96*, 1996. (Cited in Section 4.)

[135] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. *SIGOPS Operating Systems Review*, 30(5):84–92, September 1996. (Cited in Section 2.4.)

[136] K. Dan Levin. Adaptive structuring of distributed databases. In *Proceedings of the June 7-10, 1982, National Computer Conference*, AFIPS '82, page 691–696, New York, NY, USA, 1982. Association for Computing Machinery. (Cited in Section 1.5.)

[137] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012. (Cited in Sections 1.3.2 and 3.)

[138] S. Liu and M. Vukolic. Leader set selection for low-latency geo-replicated state machine. *IEEE TPDS*, 28(7):1933–1946, July 2017. (Cited in Sections 1.3.3, 4, 4.1, and 4.3.11.)

[139] G. Losa, V. Kumar, H. Andrade, B. Gedik, M. Hirzel, R. Soulé, and K.-L. Wu. Capsule: Language and system support for efficient state sharing in distributed stream processing systems, 2012. (Cited in Sections 5.1.2 and 5.1.3.)

[140] Christopher R. Lumb. Façade: Virtual storage devices with performance guarantees. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association. (Cited in Section 2.3.3.)

[141] K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine. In *Proc. of 33rd IEEE International Conference on Data Engineering (ICDE)*, pages 227–230, 2017. (Cited in Section 5.1.3.)

[142] Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing SOCC'14*, 2014. (Cited in Section 4.)

[143] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148, 2015. (Cited in Section 5.1.3.)

[144] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12), August 2017. (Cited in Sections 5, 5.1.1, and 5.1.2.)

[145] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88,

pages 8–17, New York, NY, USA, 1988. Association for Computing Machinery. (Cited in Sections 3.1, 3.2, and 4.)

[146] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4), June 1996. (Cited in Sections 1.3.2, 2, 2, 2.1, 3, and 3.1.)

[147] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, 2014. (Cited in Sections 1.3.2, 3, 3.1, and 3.2.)

[148] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC'14*, Philadelphia, PA, 2014. (Cited in Section 4.)

[149] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. Migcep: Operator migration for mobility driven distributed complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, pages 183–194, New York, NY, USA, 2013. Association for Computing Machinery. (Cited in Section 5.1.3.)

[150] Fengfeng Pan, Yinliang Yue, and Jin Xiong. Dcompaction: Delayed compaction for the lsm-tree. *Int. J. Parallel Program.*, 45(6):1310–1325, December 2017. (Cited in Section 1.3.2.)

[151] A. Papaioannou and K. Magoutis. Incremental elasticity for nosql data stores. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, 2017. (Cited in Sections 5.2.2 and 5.5.)

[152] A. Papaioannou and K. Magoutis. Replica-group leadership change as a performance enhancing mechanism in nosql data stores. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1448–1453, Los Alamitos, CA, USA, jul 2018. IEEE Computer Society. (Cited in Sections (document), 4.1, 4.1, 4.3.11, and 5.4.4.)

[153] A. Papaioannou and K. Magoutis. Replica-group leadership change as a performance enhancing mechanism in nosql data stores. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018. (Cited in Section 4.1.)

[154] Antonis Papaioannou. Linear Road Flink implementation. `https://github.com/antonis-papaioannou/linearRoad`, 2021. (Cited in Sections 4 and 5.3.)

[155] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *SIGOPS Oper. Syst. Rev.*, 27(2):21–34, April 1993. (Cited in Section 4.1.)

[156] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017. (Cited in Section 1.5.)

[157] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proc. of the 6th USENIX Conference on File and Storage Technologies, FAST'08*, San Jose, California, 2008. (Cited in Sections 1.3.1 and 2.)

[158] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, 2006. (Cited in Section 5.1.3.)

[159] Nigel Rayner, Donald Feinberg, Massimo Pezzini, and Roxane Edjlali. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. `https://www.gartner.com/imagesrv/media-products/pdf/Kx/KX-1-3CZ44RH.pdf`. (Cited in Section 1.2.)

[160] Roger Rea. Walmart & IBM Revisit the Linear Road Benchmark. `https://www.slideshare.net/RedisLabs/walmart-ibm-revisit-the-linear-road-benchmark`, May 10-11, 2016. (retrieved July 2021). (Cited in Sections 5, 5.1.1, and 5.3.2.)

[161] Reinsel, David and Gantz, John and Rydning, John. The Digitization of the World From Edge to Core. White paper, IDC, April 2017. (Cited in Section 1.1.)

[162] Y. Robert, F. Vivien, and D. Zaidouni. On the complexity of scheduling checkpoints for computational workflows. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6, 2012. (Cited in Section 5.1.3.)

[163] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM. (Cited in Section 1.3.2.)

[164] Nathan Senthil and Gedik Bugra. Using InfoSphere Streams with memcached and Redis. In *IBM developerWorks*, 2013. (Cited in Sections 5 and 5.1.1.)

[165] M. A. Shah, J. M. Hellerstein, Sirish Chandrasekaran, and M. J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 25–36, 2003. (Cited in Section 5.1.3.)

[166] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. Take me to your leader! online optimization of distributed storage configurations. *Proc. VLDB Endow.*, 8(12):1490–1501, August 2015. (Cited in Sections 1.3.3, 4, and 4.1.)

[167] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. (Cited in Section 1.1.)

[168] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proc. 2012 USENIX Annual Technical Conference (ATC 12)*, Boston, MA, 2012. (Cited in Sections 1.3.2 and 3.)

[169] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. Sap hana: The evolution from a modern main-memory data platform to an enterprise application platform. *Proc. VLDB Endow.*, 6(11):1184–1185, August 2013. (Cited in Section 1.2.)

[170] D. Stamatakis et al. A general-purpose architecture for replicated metadata services in distributed file systems. *IEEE TPDS*, 28(10):2747–2759, Oct 2017. (Cited in Section 4.1.)

[171] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, page 149–160, New York, NY, USA, 2001. Association for Computing Machinery. (Cited in Section 2.4.)

[172] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1081–1092. VLDB Endowment, 2006. (Cited in Section 1.5.)

[173] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. Using probabilistic reasoning to automate software tuning. *SIGMETRICS Perform. Eval. Rev.*, 32(1):404–405, June 2004. (Cited in Section 1.5.)

[174] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association. (Cited in Section 1.1.)

[175] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings*

of the 10th USENIX Conference on File and Storage Technologies, FAST'12, page 18, USA, 2012. USENIX Association. (Cited in Sections 2, 2.1, and 2.4.)

[176] Roshan Sumbaly, Jay Kreps, and Sam Shah. The big data ecosystem at linkedin. In *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, 2013. (Cited in Sections 5 and 5.1.1.)

[177] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association. (Cited in Sections 1.3.2, 1.3.3, and 4.1.)

[178] The Borealis Team. Borealis application programmer's guide [white paper]. `http://cs.brown.edu/research/borealis/public/publications/borealis_application_guide.pdf`. (Accessed 06/2021). (Cited in Section 5.1.)

[179] Wenhu Tian, Pat Martin, and Wendy Powley. Techniques for automatically sizing multiple buffer pools in db2. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, pages 294–302. IBM Press, 2003. (Cited in Section 1.5.)

[180] R. S. Tibbetts. *Linear Road: Benchmarking Stream-Based Data Management Systems*. PhD thesis, 2003. (Cited in Sections 1.3.4, 4, 5, 4, and 5.3.)

[181] QC. To, J. Soto, and V. Markl. A survey of state management in big data processing systems. *The VLDB Journal*, 27:847–872, 2018. (Cited in Sections 1.3.4 and 5.1.2.)

[182] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. Automated, elastic resource provisioning for nosql clusters using tiramola. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 34–41, May 2013. (Cited in Sections 1.3.1 and 2.4.)

[183] Sandeep Uttamchandani, Kaladhar Voruganti, Sudarshan Srinivasan, John Palmer, and D Peace. Polus: Growing storage qos management beyond a "four-year old kid".

In *3rd USENIX Conference on File and Storage Technologies (FAST'04)*, 2004. (Cited in Section 1.5.)

[184] Sandeep Uttamchandani, Li Yin, Guillermo A Alvarez, John Palmer, and Gul A Agha. Chameleon: A self-evolving, fully-adaptive resource arbitrator for storage systems. In *USENIX Annual Technical Conference, General Track*, pages 75–88, 2005. (Cited in Section 1.5.)

[185] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. kelley. Db2 advisor: an optimizer smart enough to recommend its own indexes. In *Proceedings 16th International Conference on Data Engineering*, pages 101–110, 2000. (Cited in Sections 1.5 and 8.)

[186] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery. (Cited in Section 1.5.)

[187] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04*, San Francisco, CA, 2004. (Cited in Sections 3.1 and 4.)

[188] Gerhard Weikum, Christof Hasse, Axel Mönkeberg, and Peter Zabback. The comfort automatic tuning project. *Information Systems*, 19(5):381–432, 1994. (Cited in Section 1.5.)

[189] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM J. TDS*, 22(2), June 1997. (Cited in Sections 1.3.3, 4, and 4.1.)

[190] S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *Proc. of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS'12)*, pages 278–289, 2012. (Cited in Section 5.1.3.)

[191] Y. Wu and K. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *Proc. of 31st IEEE International Conference on Data Engineering (ICDE'15)*, 2015. (Cited in Sections 5.1.2, 5.2.2, and 5.5.)

[192] Lianghong Xu et al. Reducing replication bandwidth for distributed document databases. In *Proc. of the 6th SoCC '15*, 2015. (Cited in Sections 4.2.1 and 4.3.10.)

[193] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, pages 37–42, New York, NY, USA, 2015. Association for Computing Machinery. (Cited in Section 1.1.)

[194] C. T. Yu, Cheing-mei Suen, K. Lam, and M. K. Siu. Adaptive record clustering. *ACM Trans. Database Syst.*, 10(2):180–204, June 1985. (Cited in Section 1.5.)

[195] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. Pipelined compaction for the lsm-tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 777–786, 2014. (Cited in Section 1.3.2.)

Appendix A

Publications

The research activity related to this thesis has so far produced the following publications (ordered by publication date):

1. Antonis Papaioannou and Kostas Magoutis, 2017. *Incremental elasticity for NoSQL data stores*. In the 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017), Atlanta, GA, USA, June 2017, (two page proceedings paper accompanying a poster)

2. Antonis Papaioannou and Kostas Magoutis, 2017. *Incremental elasticity for NoSQL data stores*. In the 36th IEEE International Symposium on Reliable Distributed Systems (SRDS 2017), Hong Kong, China, September, 2017

3. Antonis Papaioannou and Kostas Magoutis, 2018. *Replica-group leadership change as a performance enhancing mechanism in NoSQL data stores*. In the 38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018), Vienna, Austria, July, 2018

4. Aris Chronarakis, Antonis Papaioannou and Kostas Magoutis, 2018. *On the impact of log compaction on incrementally checkpointing stateful stream-processing operators*. In the 38th IEEE International Symposium on Reliable Distributed Systems Workshops (SRDSW), Lyon, France, October, 2019

5. Antonis Papaioannou, 2020. *Novel adaptation mechanisms for distributed data stores*. In the 14th EuroSys Doctoral Workshop (EuroDW 2020), Heraklion, Greece, April 2020

6. Antonis Papaioannoui, Chrysostomos Zeginis and Kostas Magoutis, 2020. *The Case for Better Integrating Scalable Data Stores and Stream-Processing Systems*. In the IEEE Cluster Conference 2020, Kobe, Japan, (two page proceedings paper accompanying a poster)

7. Antonis Papaioannou and Kostas Magoutis. *Addressing the read-performance impact of reconfigurations in replicated key-value stores*. In IEEE Transactions on Parallel and Distributed System (TPDS), (accepted)

8. Antonis Papaioannou and Kostas Magoutis, 2021. *Amoeba: Aligning Stream Processing Operators with Externally-Managed State*. In the 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2021), Leicester, UK, December, 2021