Building Scalable and Consistent Distributed Databases Under Conflicts

by

Hua Fan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Angela Demke Brown Associate Professor, University of Toronto Dept. of Computer Science
Supervisor(s):	Wojciech Golab Assistant Professor, University of Waterloo Dept. of Electrical and Computer Engineering
Internal Member:	Lin Tan Associate Professor, University of Waterloo Dept. of Electrical and Computer Engineering
Internal Member:	Derek Rayside Associate Professor, University of Waterloo Dept. of Electrical and Computer Engineering
Internal-External Member:	Bernard Wong Associate Professor, University of Waterloo David R. Cheriton School of Computer Science

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Distributed databases, which rely on redundant and distributed storage across multiple servers, are able to provide mission-critical data management services at large scale. Parallelism is the key to the scalability of distributed databases, but concurrent queries having conflicts may block or abort each other when strong consistency is enforced using rigorous concurrency control protocols. This thesis studies the techniques of building scalable distributed databases under strong consistency guarantees even in the face of high contention workloads. The techniques proposed in this thesis share a common idea, *conflict mitigation*, meaning mitigating conflicts by rescheduling operations in the concurrency control in the first place instead of resolving contending conflicts. Using this idea, concurrent queries under conflicts can be executed with high parallelism. This thesis explores this idea on both databases that support serializable ACID (atomic, consistency, isolation, durability) transactions, and eventually consistent NoSQL systems.

First, the epoch-based concurrency control (ECC) technique is proposed in ALOHA-KV, a new distributed key-value store that supports high performance read-only and write-only distributed transactions. ECC demonstrates that concurrent serializable distributed transactions can be processed in parallel with low overhead even under high contention. With ECC, a new atomic commitment protocol is developed that only requires amortized one round trip for a distributed write-only transaction to commit in the absence of failures.

Second, a novel paradigm of serializable distributed transaction processing is developed to extend ECC with read-write transaction processing support. This paradigm uses a newly proposed database operator, *functors*, which is a placeholder for the value of a key, which can be computed asynchronously in parallel with other functor computations of the same or other transactions. Functor-enabled ECC achieves more fine-grained concurrency control than transaction level concurrency control, and it never aborts transactions due to read-write or write-write conflicts but allows transactions to fail due to logic errors or constraint violations while guaranteeing serializability.

Lastly, this thesis explores consistency in the eventually consistent system, Apache Cassandra, for an investigation of the consistency violation, referred to as "consistency spikes". This investigation shows that the consistency spikes exhibited by Cassandra are strongly correlated with garbage collection, particularly the "stop-the-world" phase in the Java virtual machine. Thus, delaying read operations artificially at servers immediately after a garbage collection pause can virtually eliminate these spikes.

All together, these techniques allow distributed databases to provide scalable and consistent storage service.

Acknowledgements

I would like to thank my supervisor, Prof. Wojciech Golab, for his invaluable and constant guidance throughout my Ph.D. studies. He has been the strongest influence in shaping how I conduct research and teaching. Prof. Golab has always been supportive and inspiring, encouraging me to think precise, logical, and sharp. I deeply appreciate the remaining members of my examination committee for their comments and insightful suggestions during the entire course of my study. They are Prof. Lin Tan (internal member), Prof. Derek Rayside (internal member), Prof. Bernard Wong (internal-external member), and Prof. Angela Demke Brown (external examiner).

I am also grateful to many individuals whom I had opportunities to collaborate with and get inspiration from. Thanks to Dr. Jeff Pound of SAP labs, Waterloo, Dr. Chi Zhang and Dr. Yong Wang of Google Inc., Mountain View, for hosting such wonderful internships in their companies and inspiring me in my research. Thanks to Dr. Charles Brad Morrey III and Dr. Xiaozhou Li of Google Inc., Mountain View, for their collaboration on my research projects and pushing my research to a greater level of depth. Thanks to lab colleagues and collaborators, Aditya Ramaraju, Marlon McKenzie, and Shankha Chatterjee for their helpful suggestions and feedback.

Living and studying in Waterloo for nearly five years was a delightful and enriching journey in my life. This thesis cannot be complete without the support of my family. Special thanks to my parents, Heng Fan and Qiongying Yan, and my sister Dr. Yingjie Fan, for their endless support and encouragement. Thanks to my son, Junmao Fan, for bringing so many joyful days and nights, which motivated me to do great things.

Dedication

To my parents

Table of Contents

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 Read-only and Write-only Distributed Transactions Under Conflicts 3

1.2 Serializable Read-Write Distributed Transaction Processing 5

1.3 Investigation of Consistency Anomalies in Apache Cassandra 7

1.4 Outline and Previously Published Work 9

2 Background 10

2.1 Transactions in Databases 10

2.1.1 Concurrency Control Techniques 11

2.1.2 Transaction Isolation Levels 12

2.1.3 Serializability 13

2.2 Distributed Transactions 14

2.2.1 Atomic Commitment Protocols 14

2.2.2 Combination of 2PL and 2PC 15

2.2.3 RAMP: Read Atomic Multi-Partition Transaction 16

2.2.4 Calvin: Deterministic Databases 17

2.3 Consistency in BASE Systems 18

2.3.1 CAP theorem . . . 19
2.3.2 Eventual Consistency . . . 20
2.3.3 Consistency Analysis . . . 20
2.4 Summary . . . 22

3 Epoch-based Concurrency Control and ALOHA-KV 23
3.1 Introduction . . . 23
3.2 System Model and Architecture . . . 24
3.2.1 System Model . . . 25
3.2.2 Architecture . . . 26
3.3 Epoch-based Concurrency Control Mechanism . . . 27
3.3.1 Invariants and Rules . . . 28
3.3.2 Transaction Barriers . . . 29
3.3.3 Example . . . 29
3.4 Implementation . . . 31
3.4.1 Data Representation in Storage . . . 31
3.4.2 Transaction Protocol . . . 32
3.4.3 On the Side-Effects of Stragglers . . . 34
3.5 Fault Tolerance . . . 35
3.5.1 Replication . . . 35
3.5.2 Cluster Membership . . . 36
3.5.3 Logging and Checkpointing . . . 37
3.6 Theoretical Analysis . . . 38
3.6.1 System Throughput . . . 38
3.6.2 Epoch Switch Scalability . . . 39
3.6.3 Analysis of Serializability . . . 42
3.7 Experimental Evaluation . . . 43
3.7.1 Experimental Setup . . . 43

3.7.2 ALOHA-KV vs. RAMP Results . . . 44
3.7.3 Microbenchmark Experiments . . . 46
3.7.4 Scalability . . . 49
3.7.5 Fault Tolerance . . . 52
3.8 Summary . . . 53

4 Scalable Serializable Transaction Processing Using Functors 54
4.1 Introduction . . . 54
4.2 Architecture and Design of ALOHA-DB . . . 56
4.2.1 Architecture . . . 56
4.2.2 Unified Epochs . . . 58
4.2.3 Avoiding the Side-Effects of Stragglers . . . 59
4.2.4 Multi-version Storage . . . 60
4.3 Functors . . . 62
4.3.1 Transaction Lifecycle . . . 62
4.3.2 Functors for Read-Write Transactions . . . 64
4.3.3 Transforming a transaction to functors . . . 64
4.3.4 Functor Computing . . . 66
4.4 Implementation Details . . . 68
4.4.1 Functor Processing In ALOHA-DB . . . 69
4.4.2 Dependent Transactions . . . 71
4.4.3 Serializability . . . 73
4.5 Evaluation . . . 74
4.5.1 Experimental Setup . . . 74
4.5.2 TPC-C Experiments . . . 77
4.5.3 Microbenchmark Experiments . . . 81
4.5.4 Discussion . . . 85
4.6 Summary . . . 87

5 Understanding the Causes of Consistency Anomalies in Apache Cassandra **88**

5.1 Introduction . . . 88
5.2 Cassandra Consistency Levels . . . 89
5.3 Hypothesis . . . 90
5.4 Experiments . . . 91
5.4.1 Consistency metric . . . 91
5.4.2 Hardware and software environment . . . 92
5.4.3 Inconsistency spikes versus STW pause . . . 93
5.4.4 Smoothing the inconsistency spikes . . . 95
5.5 Summary . . . 97

6 Related Work **99**

7 Conclusion and Future Work **103**

7.1 Conclusions . . . 103
7.2 Future Research Directions . . . 104

References **108**

List of Tables

2.1 SQL-92 isolation levels defined in terms of the three phenomena. 13

4.1 Examples of some f-types and their f-argument representations in functors. 64

5.1 Comparison of consistency violation, throughput and read latency under read delay in Apache Cassandra. 97

List of Figures

1.1 Consistency spikes: when a constant workload is applied to Cassandra deployed in a private cluster, the graph shows occasional abrupt inconsistency. 8

2.1 A non-linearizable history example. 21
2.2 A non-linearizable history example shows Γ metric. 22

3.1 Illustration of the system architecture of ALOHA-KV. 27
3.2 An example to illustrate the epochs in ECC. 30
3.3 Illustration of multiversion storage and insertion in ALOHA-KV. 32
3.4 Barrier synchronization. 40
3.5 Experiment results of throughput and latency: ALOHA-KV vs. RAMP under 20ms read/write epoch duration. Throughput and latency presented using logarithmic scales. 45
3.6 Throughput and average latency experiment results of ALOHA-KV under various epoch durations. 47
3.7 Throughput experiment results of ALOHA-KV under various transaction sizes. 48
3.8 Throughput under various experiment results of ALOHA-KV read/write proportions. 49
3.9 Aggregate and per host throughput experiment results of ALOHA-KV using transaction size 4000. 50
3.10 Epoch switch time for various numbers of FE instances in experiments of ALOHA-KV. 51
3.11 Evaluation result of various fault tolerance strategies in experiments of ALOHA-KV. 52

4.1 Illustration of the system architecture of ALOHA-DB. 57
4.2 Illustration of unified epochs in ALOHA-DB. 59
4.3 Avoiding straggler side-effects by allowing transactions to start without authorization in ALOHA-DB. This figure illustrates execution under two different protocols: transactions started only with authorization (colored blue in the example) and transactions started once the previous epoch completes (colored orange in the example). 60
4.4 Illustration of the storage multi-versioning layout for one key in ALOHA-DB. The linked arrays are used to store the versions of one key. 61
4.5 Example of three transactions executed using functors over two data items. 68
4.6 Throughput vs. latency: ALOHA-DB and Calvin experiments for NewOrder transactions on eight m4.4xlarge instances. Logarithmic scale used for horizontal axis. **1W** or **10W** denotes 1 or 10 warehouses per host in TPC-C experiments; **1D** or **10D** denotes 1 or 10 districts per host in scaled TPC-C experiments. 78
4.7 ALOHA-DB and Calvin throughput for NewOrder and Payment transactions under various numbers of warehouses or districts per host. Logarithmic scale used for vertical axis. 79
4.8 ALOHA-DB and Calvin scale-out performance for NewOrder transactions. Logarithmic scales used for both axes. 80
4.9 ALOHA-DB and Calvin microbenchmark performance under various values of the contention index. Logarithmic scales used for both axes. 81
4.10 Latency breakdown: latency of different stages of a transaction lifecycle in ALOHA-DB and Calvin under low and high contentions. 83
4.11 Throughput of single-partition transactions under various transaction size. 84
4.12 ALOHA-DB and Calvin latency under various epoch durations. 85

5.1 Illustration of how STW causes severe staleness. 92
5.2 Time series of Γ score and GC pause time with five YCSB hosts and replication factor three. 94
5.3 Time series of Γ score and GC pause time with replication factor five. . . 95
5.4 Smoothing of inconsistency spikes by delaying reads artificially after a GC STW pause. 96

Chapter 1

Introduction

It is commonly believed that we are already in the "data age", where applications and their users are producing and consuming much more data than that of years before. The prevalence of faster computing devices and higher speed Internet further help build many "Big Data" applications, that challenge the designs and operations of data management systems. Those applications prefer distributed data management systems, which distribute workloads to multiple servers, for performance, data safety, and availability reasons. Specifically, distributed databases can scale out to more servers to add storage and processing capacity of the system. Replication, a technique widely used in distributed systems for fault tolerance, allows the whole system to retain all data even if some server fails. Replicas also help the system to improve throughput by sharing the workload; clients may contact the closest replica for better latency.

Scalable distributed database systems, however, are notoriously hard to design. Concurrent queries are executed using distributed protocols that ensure crucial correctness properties in a highly concurrent, failure-prone environment. Violation of those correctness properties may result in an inconsistent state of the system. The parallelism is crucial for scalable performance, but coordination and synchronization overhead in those protocols may hurt the parallelism of the system. Especially, distributed databases in high contention workloads face scalability and consistency challenges from conflicts. Concurrent queries have conflicts when they intend to access a common data item, and one of the queries intends to update the data item. They may block or abort each other by the action of the concurrency control mechanism, without which the systems may be subject to consistency violations.

This thesis focuses on techniques for building scalable distributed databases that provide

critical consistency guarantees, even when the systems are under conflicts from concurrent requests. In particular, read-write conflicts and write-write conflicts challenge the scalability of distributed transactions with serializable isolation and atomicity guarantees, and read-write conflicts may result in undesirable consistency anomalies in NoSQL storage systems that only have a weakly consistent guarantee. This thesis presents 1) the design and prototype of two systems that provide high performance distributed transactions with serializability guarantees, and 2) the investigation and improvement of the consistency of a scalable NoSQL storage system as follows:

1. **ALOHA-KV** is a distributed in-memory key-value store that supports high performance *read-only* and *write-only* transactions, which are also known as *multi-put* and *multi-get* operations. ALOHA-KV can process transactions in parallel even under high contention cases using an epoch-based concurrency control protocol proposed by this thesis. ALOHA-KV also uses a novel atomic commitment protocol that only requires amortized one round trip for write-only transactions in failure-free cases.

2. **ALOHA-DB** is a transaction processing layer on top of ALOHA-KV, and it supports high performance serializable distributed read-write transactions. ALOHA-DB allows contended transactions to be processed in parallel using fine-grained concurrency control. In experimental results, ALOHA-DB provides one to two orders of magnitude higher performance compared to the deterministic database Calvin, which is the current promising distributed transaction solution.

3. **An investigation** of the consistency anomalies in Apache Cassandra, an open source distributed database, is presented in this thesis. The investigation explains the cause of the phenomenon referred to as "consistency spikes", and proposes an efficient way to virtually eliminate the spikes.

The key idea behind these three works is *conflict mitigation*, which mitigates potential conflicts by rescheduling operations for overall scalability and consistency. Conventionally, concurrent operation requests may try their best to contend for some shared resource. However, contention is the real hurdle for throughput, latency, and consistency for the systems. For example, serializable distributed transactions are known to have scalability problems [12, 14, 22, 19, 52, 101], because the concurrency controls and commitment protocols for these transactions suffer under conflicts. On the other side of the consistency spectrum, NoSQL systems can easily achieve linear scalability by trading off consistency guarantees, but are vulnerable to consistency anomalies in the presence of conflicts [18, 67].

To build scalable and consistent databases, this thesis takes a simple but effective perspective: reschedule the execution of potentially conflicting operations to avoid potential conflicts. Figuratively, each concurrent operation may pay extra cost on latency for "politeness" — avoiding potential conflicts in the first place, but the systems gain overall throughput and consistency, and even latency, because operations can be processed highly in parallel and optimistically without concerns about consistency.

1.1 Read-only and Write-only Distributed Transactions Under Conflicts

To overcome the performance gap between traditional relational databases and web-scale data requirements, many distributed storage systems have forsaken strong transactions in favor of simplified but more scalable design in some mission-critical systems, including shopping carts [32] and social network storage [97]. Although many of these NoSQL systems do provide high-level query languages, they do not support the relational model fully and instead offer a much narrower API paired with weak consistency. Several forces drive these design choices: the need for a more flexible schema to accommodate a broader variety of data sets, the inherent trade-off between consistency and availability during a network partition [24], and the inherent trade-off between consistency and latency [6]. While NoSQL systems attract intensive attention from researchers and are widely deployed in industry, an alternative school of thought favors the continued use of transactions and strong consistency in distributed storage systems [8, 9, 14, 29].

Transaction processing [21, 50] technology is the key to the coherent data management and reliable information exploitation. Transactions in database systems must access or update related data items together to correctly reflect the real-world phenomena and activities. In each transaction, any interruption or interleaving of updates and accesses from other transactions may make the data inconsistent. The *ACID* semantics: *A*tomicity, *C*onsistency, *I*solation and *D*urability, are referred to as the golden standard of transactions in database systems.

Concurrent transactions have *conflicts* when at least one of the transactions intends to modify a common data item. Concurrency control mechanisms must be applied to resolve the conflicts and to provide required transaction isolation levels. Concurrency control mechanism designs play a key role in the consistency and scalability of the systems [20, 21, 52, 55]: concurrency control mechanisms maintain consistent states of the databases systems by providing an illusion of isolated execution, and they may impact the database system

performance by blocking the execution of conflicting transactions.

Recent research efforts have explored coordination avoidance in special cases [12, 82], as well as the more general trade-off between transaction isolation and performance under the following assumption: concurrent serializable transactions under read-write or write-write conflicts require costly synchronization, and thus may incur a steep price in terms of performance [12]. However, this assumption ignores the possibility that conflicting writes need not block each other, or violate serializability if read-write conflicts are not present at the same time. This thesis presents serializable read-only and write-only distributed transactions as a counterexample to show that concurrent transactions can be processed in parallel with low overhead despite conflicts.

Atomic read-only or write-only transactions, although less powerful than general ACID transactions, have been debated intensely in recent research [30, 39, 65, 85]. They are well suited to systems that process reads and writes in batches for efficiency, but also require atomicity for each batch (i.e., two writes within one batch must both succeed or both fail). For example, StoreAll with Express Query [54] is a scalable file metadata system built on top of LazyBase, a distributed storage layer in which clients batch updates together into *self-contained units* (SCUs) [26]. Write-only transactions naturally support the atomic insertion of an SCU containing up to thousands of updates. For instance, atomically moving a set of files from system A to system B, involves deleting metadata in system A and inserting in system B. Another application of such transactions is distributed system automatic reconfiguration [83] to achieve data migration and dynamic replication factor adjustment [75, 88]. In addition, Chapter 4 discusses the possibility of using atomic read-only or write-only transactions to support high-performance read-write transactions processing.

While useful in practice, read-only or write-only transactions in distributed storage systems are considered inherently costly. They require concurrency control for transaction isolation since both read-write and write-write conflicts are possible, and they rely on distributed commitment protocols to ensure atomicity in the presence of failures. Many systems either bite the bullet and pay a performance penalty for serializable distributed transactions [9, 29, 87], or sacrifice serializability while providing some alternative form of strong consistency [14, 59]. The transactional solutions rely on costly atomic commitment protocols in the sense that at least two network round trips are required to commit a transaction in the presence of contention. The high contention footprint of such protocols [70, 93] easily becomes a performance bottleneck as contention increases, which triggers additional protocol messages as conflicting transactions resolve their relative serialization order.

In search of a simpler and leaner protocol for read-only/write-only(ro/wo) transactions,

this thesis presents a system that draws inspiration from the slotted ALOHA network protocol [81]. Specifically, this thesis proposes a scheme for supporting serializable multi-partition read-only and write-only transactions by splitting time into read-only and write-only epochs. This design relies on an *epoch-based concurrency control* (ECC) mechanism that minimizes conflicts between multi-partition transactions while using minimal metadata, thus enabling high throughput. To better understand the performance envelope of ECC, this work incorporates it into a scalable distributed key-value store, called ALOHA-KV, and compares it against RAMP [14], a scalable distributed transaction protocol that can only provide a weak isolation level. The experimental results show that when transaction size exceeds 10 key-value pairs, ALOHA-KV outperforms RAMP up to three orders of magnitude in terms of both throughput *and* latency, at the same time providing stronger transaction isolation.

1.2 Serializable Read-Write Distributed Transaction Processing

Epoch-based concurrency control combines multi-versioning (MV) and timestamp ordering (TSO), two non-blocking concurrency control techniques, and makes the effects of transactions visible at epoch boundaries. It is presented in ALOHA-KV, a system that supports high performance read-only and write-only transactions. It seeks to avoid most forms of coordination between transactions by keeping reads and writes completely separated in time. However, this idea has a clear difficulty to overcome: the common case of a single transaction that does both reading and writing.

Unfortunately, the cost of serializable read-write transactions is high in a distributed environment, where conventional techniques fare poorly especially for update-intensive workloads [52, 80]. In particular, two-phase locking leads to aborts due to (suspected or actual) deadlock, optimistic protocols suffer from aborts due to contention, and two-phase commit increases the abort rate further by enlarging the "contention footprint" of a transaction [70, 93]. These conventional techniques take a transaction as the basic unit of concurrency control, meaning that a transaction can only commit keys after all conflicts for these keys have been resolved by holding locks or completing backward validation in optimistic protocols. This thesis refers to these techniques as *transaction-level* concurrency control.

Recently, a few research works [80, 92, 93] aim to boost performance for serializable distributed transactions under contention by enforcing *deterministic* transaction execution

scheduling on all partitions to prevent aborts, which avoids wasted work due to transaction restarts but introduces its own overhead. By assuming all involved partitions will successfully execute transactions according to the same deterministic scheduling, a transaction is able to commit keys for one partition as long as all conflicts for these keys have been resolved, irrespective of any unresolved conflict on other partitions. This thesis refers to these methods as *partition-level* concurrency control, which gains more parallelism than the transaction-level scheme under highly contended distributed transaction workloads.

This thesis proposes a novel paradigm of serializable transaction processing using *functors*, which conceptually resemble *futures* [63] in modern programming languages. A functor is a placeholder for the value of a key, and this value can be computed *asynchronously* in the future *in parallel* with other functor computations. Functors enable more fine-grained concurrency control than in partition-level or transaction-level schemes, and distribution of work for a given transaction that tends to co-locate computation with storage. *First*, functor computing only focuses on how to compute the value of a key, thus only requires *key-level* concurrency control (i.e., resolve conflicts for generating the value of the functor's key) when ECC already guarantees transaction atomicity and resolves transaction orders. *Second*, a functor is computed on the host where the functor's key is stored. Thus the transaction processing overhead can be distributed and offloaded to all the participant partitions. The functor paradigm focuses on the transaction model of server-side stored procedures, and introduces extra latency due to the nature of asynchronous processing. However, as the experimental results in this thesis show, the benefit of conflict mitigation using the functor paradigm surpasses the overhead of asynchronous processing, resulting in overall lower latency.

Functors elevate epoch-based concurrency control to a new level: supporting serializable distributed read-write transactions. In functor-enabled ECC, a read-write transaction is executed in two phases: a *write-only phase* that uses a write-only transaction to store a collection of *functors* under the low overhead of ECC, and a *computing phase* that determines the outcomes of the functors asynchronously. With multi-versioning in ECC, the functor computing only relies on accessing historical versions. Thus, the traditional locking mechanism is not needed to read a historical version. This combination of techniques never aborts transactions due to read-write or write-write conflicts and yet allows transactions to fail due to logic errors or constraint violations while guaranteeing serializability.

The high-level idea of executing read-write transactions on top of a blind write layer (write-only transactions) was previously explored in some scale-out distributed databases [19, 22, 46]. For example, Hyder [22] is a log-structured database that executes transactions optimistically using a snapshot version, and appends writes as "intentions" atomically to the log. Then, a centralized validation phase is needed to decide if the transaction

should be committed or aborted. The processing of functors in this thesis differs from the "melding" procedure in Hyder [22] in several ways: functors are placeholders for values whereas "intentions" in Hyder record concrete values; functors are evaluated using the latest version of the data as opposed to a slightly stale snapshot; and the computation in functor-enabled ECC is partitioned across servers.

The functor-enabled ECC is implemented and incorporated with ALOHA-KV into a new scalable distributed transaction processing system called ALOHA-DB. On the TPC-C benchmark for distributed read-write transactions and a YCSB-like microbenchmark, ALOHA-DB outperforms Calvin [80, 92, 93] – a state-of-the-art high-performance distributed transaction system that uses deterministic scheduling – in throughput, while also maintaining lower latency. Furthermore, ALOHA-DB allows transactions to abort due to logic errors, as required by the benchmark.

1.3 Investigation of Consistency Anomalies in Apache Cassandra

Different from ALOHA-KV and ALOHA-DB, many NoSQL systems can easily achieve linear scalability by giving up transactions and strong consistency support, which may have high overhead as explained in Section 1.1. The justification of NoSQL systems is backed by the wide acceptance of *CAP theorem*. The *CAP theorem*, conjectured by Brewer [24] in 2000, and formally proved by Gilbert and Lynch [45] in 2002, states that any distributed system cannot simultaneously provide: *C*onsistency, *A*vailability and *P*artition tolerance. For large scale systems, network partitions are inevitable, and those systems choose weak consistency to support high availability. Compromising consistency in favor of availability, NoSQL systems also adopt the simplicity of design and achieve scalability close to linear. These systems [18, 32, 61, 97] often provide *BASE* (*B*asically *A*vailable, *S*oft state, *E*ventual consistency) semantics [43].

An interesting phenomenon was observed on benchmarking an eventually consistent system [77], Apache Cassandra: when a constant workload is applied on Cassandra, and an inconsistency metric that measures the inconsistency in time unit is plotted in a time series graph, the graph shows occasional abrupt inconsistency. Figure 1.1 is a copy of Figure 3 in [77][1]. These spikes in the graph are referred to as consistency spikes (staleness spikes). Similar anomalies have been observed by [18, 77, 98], leading to speculation regarding possible causes including side effects of caching and Distributed Denial of Service (DDoS)

[1]Permission granted from the copyright holder.

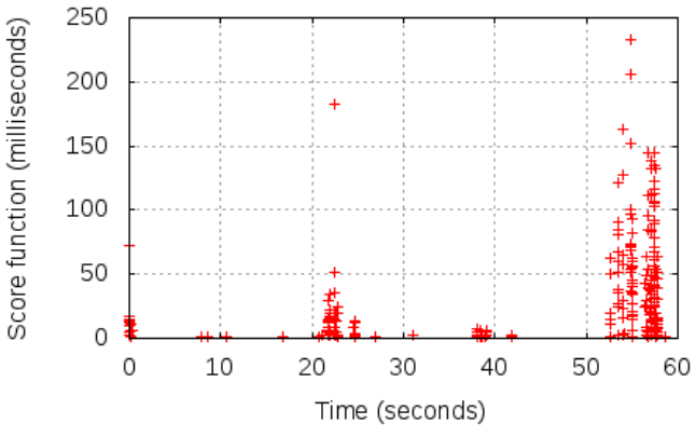


Figure 1.1: Consistency spikes: when a constant workload is applied to Cassandra deployed in a private cluster, the graph shows occasional abrupt inconsistency.

countermeasures, as well as network jitter. However, the experiments in [77] and some other experiments that follow the same methodology, are run in a single private data center, which isolates the storage system from the effects of caching layers, DDoS countermeasures, and ambient network traffic. None of the network delay mechanisms suggested above account adequately for staleness spikes in such a controlled environment.

Other than improving scalability for systems supporting strong consistency, this investigation of the consistency spikes is motivated from a different angle: understanding cause of the inconsistency, when the NoSQL system already has scalability. In particular, this investigation works around these three questions:

- Are the consistency spikes reproducible in a given private cluster environment?
- Is the processing delay the cause of the spikes? Especially, it is interesting to explore the garbage collection "stop-the-world" phase in Java virtual machine (JVM), which will pause all application threads.
- Can the spikes be smoothed out, and if so then at what cost?

The experimental results show that the garbage collection in the JVM, in particular, the "stop-the-world" phases which pause all application threads, has a high correlation with the consistency spikes. Furthermore, delaying read operations artificially at server side immediately after garbage collection, in order to avoid potential read-write conflicts, can

virtually eliminate the spikes. In the experiments, more than 98% of consistency anomalies that exceed a minimum threshold of 5ms are removed with little impact on throughput and latency.

1.4 Outline and Previously Published Work

The remainder of this thesis is organized as the follows: Chapter 2 presents the background and literature survey of transaction concurrency control mechanism, eventual consistency, and consistency analysis theories. The epoch-based concurrency control mechanism for distributed read-only and write-only transactions is given in Chapter 3. The design and implementation of ALOHA-KV is presented in the same chapter. In Chapter 4, functor and functor-enabled ECC are discussed, for supporting scalable distributed read-write transactions, and the high-performance transaction processing system ALOHA-DB is proposed. Chapter 5 discusses the investigation of understanding and eliminating consistency spikes. Related work of this thesis is discussed in Chapter 6. Finally, the thesis concludes and highlights future work in Chapter 7.

Chapter 1 includes material from previous publications [39, 41, 42]. Chapter 3 is derived material from [39, 41]. Chapter 4 is derived material from a paper co-authored with Dr. Golab accepted by ICDCS 2018. Chapter 5 is derived material from [42].

Chapter 2

Background

This chapter presents the background and some prior works related to this thesis. Section 2.1 begins with the transaction concepts with ACID properties, then discusses concurrency control mechanisms and the transaction isolation. Section 2.2 discusses distributed transactions, and describes various distributed transaction protocols. In Section 2.3, the background and theory of analysis of eventual consistency is presented.

2.1 Transactions in Databases

The concept of transactions in database systems provides a very convenient and powerful interface for application developers. In particular, transactions implement the ACID properties, which denotes the following:

- Atomicity. The transaction takes effect in an "all or nothing" fashion. If part of the transaction fails, the entire transaction fails, and the database is left unchanged.
- Consistency. The transaction must preserve the data integrity constraints of the database, meaning the transaction brings the database from one valid state to another.
- Isolation. Transaction isolation provides the illusion as if the transaction accesses the database alone, even in the presence of other transactions.
- Durability. Once a transaction is committed, the transaction's actions must persist across crashes.

Violation of either atomicity or isolation may leave the database in an inconsistent state. This subsection will discuss the technique of concurrency control and transaction isolation, which are crucial to the performance. Section 2.2.1 will present a protocol to provide atomicity for distributed transactions.

2.1.1 Concurrency Control Techniques

Transaction execution parallelism is the key to the performance of database systems. Concurrency control mechanisms are used to implement the transaction isolation requirement in databases. How to resolve conflicts is the key to a concurrency control mechanism. The following lists the common concurrency control techniques used in databases:

- **Locking**. These concurrency control mechanisms use locks to synchronize concurrent transactions for accessing shared data items. Lock-based concurrency control mechanisms vary in aspects such as locking durations [7], conflict resolution methods (e.g., blocking or aborting) and deadlock avoidance techniques, etc. *Two-phase locking (2PL)* [35] is a standard concurrency control mechanism for serializable isolation (detailed later in this section). Informally, using 2PL, a transaction must obtain any locks it needs before it releases any locks. It includes two phases, the growing phase when the transaction acquires all locks required for the transaction and never releases any locks, and the shrinking phase when the transaction releases the acquired locks and never acquires any locks. Thus in high contention workloads, 2PL may require a transaction to wait a significant time for a lock. Lock-based concurrency control is a *pessimistic* concurrency control method because it allows transaction execution only after all conflicts have been resolved by locking.

- **Optimistic Concurrency Control (OCC)**. OCC [60] executes transactions speculatively without the requirement of holding locks for any data item because it assumes the conflicts are infrequent. However, concurrent transactions may have conflicts at the time of execution. OCC can resolve the conflicts at transaction commit time using backward validation to guarantee the required isolation level. If a transaction fails the backward validation, OCC must abort the transaction. Thus, OCC leads to high abort rates under high contention, which has relatively high overhead because the validation is at the end of the execution when the failed transaction has already consumed the resource (CPU, I/O, network, etc.) of the transaction processing systems.

- **Multiversion Concurrency Control (MVCC)**. Multiversioning (MV) allows a data item to keep more than one version in the concurrency control mechanism.

With MV, the committed versions are always available for reading, even if another transaction is simultaneously writing the data (for another version). MVCC uses MV for concurrency control, and the isolation level decides which version is visible for each transaction. The most common isolation level implemented by MVCC is *snapshot isolation (SI)*, where the transactions are allowed to be executed based on a committed version. MVCC never blocks read operations but requires extra overhead to create and maintain the versions.

- **Timestamp Ordering (TSO)**. The TSO [20] protocol is a non-blocking concurrency control mechanism. In timestamp ordering, a transaction is assigned a unique timestamp at the beginning of the transaction execution. The concurrency control uses the timestamps to decide the transaction execution order. For example, in a serializable system using TSO, if a transaction T_i has been assigned timestamp $TS(T_i)$, and a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$. If both transactions are committed, the system must ensure that transaction T_i appears to be executed before transaction T_j.

2.1.2 Transaction Isolation Levels

If all transactions are executed one after another, the storage system can easily enforce consistency and isolation properties. However, most current systems allow transaction execution concurrently, for the following reasons:

- Improving throughput. For example, concurrent transactions can run on different cores on the same server or different servers simultaneously, and one transaction can use CPU resources while another transaction is blocked on I/O.
- Reducing waiting time. Consider a workload comprised of long transactions and short transactions. If transactions run serially, a short transaction may need to wait until a preceding long transaction completes.

However, the consistency of the system may be violated if transactions interleave with each other. The transaction isolation property describes the level of how one transaction can influence other transactions. The standard organization ANSI formally proposed an SQL isolation level standard in 1992, which is referred to as SQL-92 [17, 69]. This standard defined three phenomena which may lead to anomalous (non-serializable) behavior and the SQL-92 isolation levels are defined by the absence of these three phenomena, as shown in Table 2.1. The following is the definition of the phenomena and isolation level quoted from reference [17]:

- **P1 (Dirty Read)**: Transaction T1 modifies a data item. Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK. If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed.
- **P2 (Non-repeatable or Fuzzy Read)**: Transaction T1 reads a data item. Another transaction T2 then modifies or deletes that data item and commits. If T1 then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.
- **P3 (Phantom)**: Transaction T1 reads a set of data items satisfying some *search condition*. Transaction T2 then creates data items that satisfy T1's search condition and commits. If T1 then repeats its read with the same search condition, it gets a set of data items different from the first read.

Table 2.1: SQL-92 isolation levels defined in terms of the three phenomena.

Isolation Level	Dirty Read	Fuzzy Read	Phantom
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

In SQL-92, serializability is the transaction isolation level that precludes all three types of isolation anomalies: phantom reads, non-repeatable reads, and dirty reads. Even though many database products refer to SQL-92 for their transaction isolation, the standard itself has been criticized for the ambiguous definition of isolation anomalies and inability to cover all possible anomalous behaviors [17]. Recently, many weaker isolation levels have been proposed [14, 31, 64, 65], but serializability is regarded as a standard of strong consistency.

2.1.3 Serializability

Informally speaking, *serializability* requires that transactions appear to take effect in a serial order even if they are executed concurrently. Serializability is widely regarded as the gold standard of transaction isolation in conventional databases. However, the overhead in concurrency control for providing serializability is commonly regarded as costly [12, 14, 22, 19, 52, 101]. For example, a survey in [13] shows that many commercial

systems do not support serializability, or use weaker isolation as the default setting to improve performance. Formal definitions of serializability are phrased in reference to histories or schedules (defined later in this subsection), which record the actions of transactions, and equivalence, which gives a precise meaning to the order in which transactions "appear to take effect" [21]. Different interpretations of equivalence give rise to different correctness properties, notably *conflict-serializability* and *view-serializability*.

An *operation*, such as read or write on a specific data item, may change the database from one state to another state. A *transaction* comprises a sequence of operations as a linear ordering of its actions. A *history* is a sequence of transactions. If a transaction only has read and write operations, two operations *conflict* if at least one of them is a write operation and they are performed by distinct transactions on the same data item. Two histories are *conflict-equivalent* if they contain the same transactions and operations, and they order conflicting operations of non-aborted transactions in the same way. A history is *conflict-serializable* (CS) if it is conflict-equivalent to some serial history. The *View equivalence* relation over histories is defined as follows: histories H and H' are *view-equivalent* if (1) they contain the same transactions and operations, (2) they have the same reads-from relationships (i.e., if T reads a value written by T' in H then the same holds for the corresponding read of T in H'), and (3) they have the same final writes (i.e., if T is the last transaction to write a data item x in H then the same holds in H'). A history is *view serializable* if it is view-equivalent to some serial history.

Serializability takes on a slightly different consideration when a database implements multi-version storage. A *multi-version (MV) history* of such a system records the specific data item version accessed by each step. A serial MV history is called *one-copy serial* if for all i, j, and x, if transaction T_i reads item x from transaction T_j, then $i = j$, or T_j is the last transaction preceding T_i that writes into any version of x. Equivalence for two MV histories H and H' is defined similarly to view-equivalence. Condition (3) follows trivially because each version is written at most once and hence each write is final. An MV history is *one-copy serializable (1SR)* if it is equivalent to a one-copy serial MV history, which means that the system behaves as though it maintained only one version of each data item.

2.2 Distributed Transactions

2.2.1 Atomic Commitment Protocols

In a non-distributed system, a failure is an all-or-nothing affair (e.g., a process crashes). However, in distributed systems, partial failures are possible. For example, some hosts are

working while others have failed. Thus, distributed databases, which may have transactions across multiple hosts, need an *atomic commitment protocol* to guarantee the transaction atomicity even in failure cases. In other words, all the participants of the transaction should agree on the transaction decision — commit or abort, even if some of them fail.

The two-phase commit (2PC) protocol [21] is a common atomic commitment protocol used in distributed systems. The 2PC protocol includes a prepare phase followed by a commit phase.

- Prepare phase (also called voting phase). A coordinator attempts to prepare all the transaction's participating processes to take the necessary steps for either committing or aborting the transaction. Participants have to vote to "commit" or "abort" the transaction. 2PC assumes a synchronous model of computation by using timeouts to detect failures.
- Commit phase. The coordinator decides on "commit" (only if all vote "commit") or "abort" (otherwise), and informs the result to all the participating processes.

Failure of the coordinator in 2PC may cause the protocol to stall—a problem addressed by three-phase commit and Paxos commit at the cost of additional rounds of communication or additional messages in the failure-free case [49, 84].

2.2.2 Combination of 2PL and 2PC

The concurrency control mechanism two-phase locking (2PL) and the atomic commitment protocol two-phase commit (2PC) both need two phases. The combination of 2PL and 2PC is a classic solution for the distributed transaction in which a transaction can be committed in two round trips with both serializable isolation and atomicity. However, concurrency control protocols have potentially higher deadlock avoidance overhead in distributed systems, because the deadlock detection or prevention decisions may need to wait for responses from remote hosts. Thus, this concurrency control technique must consider deadlock detection and avoidance in the protocol design.

Sinfonia [9] uses the combination of 2PL and a two-phase protocol modified from 2PC for serializability. In Sinfonia, a transaction acquires all the locks of needed items in the first phase. In order to prevent deadlock, Sinfonia applies a simple deadlock avoidance method: the locks are acquired by a specified order and the transaction will abort and release all locks if attempting locking fails. In the second phase, the transaction will apply the commitment

decision (commit or abort), then release all the locks. Furthermore, Yesquel [8] uses similar techniques but also incorporates timestamp ordering for multiversioning.

The atomic commitment protocol, 2PC, has an intrinsic overhead – requiring two rounds of messages even for failure-free cases. It may further enlarge the performance bottleneck of 2PL under contention: a transaction may need to wait longer for a lock when multiple rounds of remote messages are involved.

2.2.3 RAMP: Read Atomic Multi-Partition Transaction

Some systems choose isolation levels that are weaker than serializability for better performance because serializable distributed transactions have known scalability problems, which is rare in weakly consistent systems. Bailis et al. [14] proposed the criteria for scalable distributed transactions: a transaction protocol can provide scalability if it has the following two properties:

- *Synchronization independence.* That is, one transaction cannot cause another transaction to stall or fail.
- *Partition independence.* A transaction only contacts the partitions referred to by the transaction.

Recently, a weak isolation model *Read Atomic (RA)* is proposed in *Read Atomic Multi-Partition (RAMP)* [14] transaction protocols, which provides both of these properties. RA guarantees that either all or none of each transaction's updates are observed by another transaction. RAMP focuses on read-only write-only transactions, while other types of transactions can be built on top of RAMP. It uses transaction meta-data to avoid RA violations. RAMP provides an isolation level weaker than serializability. The RAMP paper [14] introduces three variations of the RAMP protocol. The remaining material in this subsection describes the basic RAMP protocol (RAMP-Fast), and gives a comparison of different variations of the RAMP protocol.

RAMP-Fast. RAMP uses multiversioning in concurrency control and chooses the largest version as the latest one. A write-only transaction will create a new version for each object to write, and the version number is assigned by the client's local clock. In the protocol of RAMP, these versions are not required to be inserted in the order of the version numbers, as RAMP does not provide serializability. Thus, inserting these versions will never be blocked or aborted (synchronization independence). In order to provide RA isolation,

the write-only transaction needs two round trips in the protocol: one round trip to store invisible data versions, the second round trip to commit these versions and make them visible to other read transactions. The data versions include a transaction meta-data — the information of the transaction that allows clients to detect RA violations. The meta-data of RAMP-Fast includes the *write-set* of the transactions. A read-only transaction needs two round trips in the presence of conflicts: one round to retrieve referred versions, the second round of reading (called *read repair*) may be issued if a non-RA read result is detected. A client can detect the RA violation by comparing version numbers from metadata that is stored and retrieved with versions.

The three variations of RAMP protocols are *RAMP-Fast*, *RAMP-Small*, and *RAMP-Hybrid* [14]. They all allow concurrent read-only and write-only operations to be executed under the above two independence properties, thus achieving scalability. However, these variations provide trade-offs between meta-data size and the number of round trips needed in the protocols. The meta-data size impacts storage and message transferring overhead; and the number of round-trips in the protocol impacts the system performance. RAMP-Fast stores the whole write-set and the transaction version number as the metadata. A read transaction can detect a RA violation when a transaction is only partially visible (e.g., data exists in meta-data's write-set, but has not been retrieved by the client). RAMP-Small stores only a write transaction's version number as metadata, but a read transaction requires an extra round of messaging to detect and resolve RA violations. In particular, the read will send all retrieved visible version numbers to all partitions participating in the transaction in order to retrieve the version number that matches an invisible version. A third algorithm is a hybrid of the above two, which also achieves performance between the other two in various experimental scenarios.

2.2.4 Calvin: Deterministic Databases

Calvin [93, 92] is a transaction scheduling and data replication layer for transaction processing in distributed databases. Calvin uses a deterministic transaction execution ordering to reduce the cost of distributed transaction concurrency control. The deterministic scheduling assumes all database partitions execute the transaction in the same order, thus no transaction may be aborted due to contention, and so Calvin does not need atomic commitment protocol (e.g., 2PC) to coordinate partitions participating in a transaction.

A transaction is duplicated to all participating partitions in Calvin. In each partition, Calvin executes the transactions by the enforced deterministic order, thus the execution on this partition does not require locking objects on a remote partition because the remote

partition has a duplicated execution with the same deterministic order. However, a partition is not able to abort a transaction arbitrarily in the protocol, for example, when the partition is overloaded or due to logic errors in the transaction.

The architecture of Calvin comprises shared-nothing partitions. Each partition is a server, including a *sequencer* and a *scheduler*, which are crucial to the replicated and deterministic scheduling. Each sequencer accepts transaction requests from clients and sends them to schedulers in a batch. Each scheduler schedules all transactions received from sequencers to a deterministic order for execution. In order to create the same transaction ordering across all schedulers, Calvin enforces that each scheduler must collect all transactions created by sequencers to form the transaction ordering in the pre-processing phase. This pre-processing phase consumes significant time, thus increasing the overall transaction latencies.

The open source implementation of Calvin [79] uses multi-threading to increase execution concurrency in each partition. However, each partition has a single-threaded lock manager to synchronize these threads. Various evaluation results [52, 80] demonstrate that Calvin surpasses both OCC or lock-based concurrency control on throughput in various high contention distributed transaction experiments, because the deterministic transaction execution simplifies the deadlock avoidance techniques and transaction commitment protocol, and avoids the high overhead conflict resolution across all participant partitions. However, a previous study [80] shows that, compared with non-deterministic databases, Calvin pays the cost at extra overhead in latency in the pre-processing phase and inability to abort transactions arbitrarily due to the deterministic execution.

2.3 Consistency in BASE Systems

Emerging Web services have introduced many challenges for distributed databases: (1) the data are replicated and distributed across a network; (2) latency and availability are crucial for websites' revenue; (3) processing simple queries, such as read or write a data item, at high speed is more crucial than processing complicated transactions. Many of the systems designed for those applications give up the support of general ACID transactions, and instead adopt *BASE* semantics [43] (*B*asically *A*vailable, *S*oft state, *E*ventual consistency). In 1997, Fox et al. [43] defined BASE semantics as follows:

- **B**asically **A**vailable. Any non-failing server will respond to a request, even if the data may be in an inconsistent or changing state.
- **S**oft states. The state of data can be regenerated at the expense of additional computation, such as comparing versions.

- **E**ventual consistency. The definition given by Doug Terry et al. states "All replicas eventually receive all writes (assuming sufficient network connectivity and reasonable reconciliation schedules), and any two replicas that have received the same set of writes have identical databases." [90]

In particular, these systems use a weaker form of concurrency control for higher performance, resulting in weaker consistency guarantees. This section will discuss these concepts and focus on eventually consistent systems.

2.3.1 CAP theorem

The CAP theorem, conjectured by Brewer et al. [24] and formalized by Gilbert and Lynch [45], targets distributed systems, such as web servers. Brewer's conjecture states that a distributed system cannot simultaneously provide all of the following three properties:

- *Consistency.* Informally, when operations are applied on a replicated object, the operations should act as if the operations are applied on a single *atomic data object* [45]. For read and write operations, any reads that begin after a write operation completes, shall return that value, or the result of a later write operation.
- *Availability.* Any non-failing server must respond with a result for every request.
- *Partition tolerance.* A *network partition* happens between two sets of hosts, when all messages are lost between hosts in one set and hosts in another set. Partition-tolerant and consistent systems (CP systems) guarantee that every response will return a value that never breaks consistency, even if any message may be lost in the network. Partition-tolerant and available systems (AP systems) guarantee that even if any message may be lost in the network, every non-failing host can respond to requests.

Based on the CAP theorem, in the presence of network partition, distributed systems need to choose between consistency and availability because at most two out of the three properties can be held. However, in the absence of network partition, distributed systems need to choose between consistency and (low) latency [6].

2.3.2 Eventual Consistency

The CAP theorem provides justification of weak consistency and asynchronous replication for "Internet scale" applications where stronger consistency has undesirable high cost and network partitions are possible. The preference of weaker consistency has gained triumph in acceptance by the industry in recent years. For example, the distributed file system GFS [44] provides atomic mutation on file namespace, but updates are applied asynchronously, and clients may read stale values; Yahoo!'s PNUTS [27] only provides per-key sequential consistency (record-level timeline consistency); COPS [64] provides another weak consistency, causal consistency. HBase [3] implements a timeline consistency: data is held on primary and replicas; updates can only be written to primary; either primary or replicas can serve read requests; data in replicas may be stale, but all replicas receive updates in the same order.

Eventual consistency is widely used in many systems, including Amazon's Dynamo [32] and its derivative versions Cassandra [61] and LinkedIn's Voldemort [4]. The eventually consistent systems allow read and write operations to continue even during network partitions, and use asynchronous replication. The update conflicts are typically managed by specialized conflict resolution procedures. The semantics provided by eventually consistent systems are far different from ACID, and are referred to as BASE semantics.

Eventually consistent systems allow inconsistency when there is a network partition. However, after the partition is healed, the system can converge to a consistent state by additional communication and computation.

2.3.3 Consistency Analysis

Eventual Consistency only guarantees the data will eventually be updated to the latest value if there are no new updates. However, it does not answer the questions of "how soon" the data will be consistent and "how stale" the data is read. This subsection discusses the consistency property linearizability and a recently proposed time-based metric Γ.

Consistency standard: Linearizability

To describe the consistency of shared memory, Lamport uses the notion of the *atomic register* to capture the behavior of shared objects for reading and writing operations [62]. Herlihy and Wing defined the linearizability [53] property, which is a correctness property for arbitrary object types. Linearizability provides the illusion that each operation applied

by concurrent processes takes effect instantaneously at some point in time between its invocation and response. This point is referred to as a linearization point. A read operation should return the value assigned by the most recent write, regardless of which process issues the operation, and all subsequent read operations should return the same result until the next write takes effect. The most recent operation and subsequent operations are determined by the linearization point.

Herlihy and Wing formalize linearizability in terms of general histories of events. Following the previous work [48], this thesis only considers histories that are well defined as in [48] for Γ metric computing. A *history* is a sequence of completed operations (already finished operations), which includes operation type, operation parameter, operation result, and operation start/finish timestamps. For a read/write operations history, if there is an explanation of linearization points for each operation, this history is linearizable. Figure 2.1 shows an example of a history of four operations applied on one data item. The operation type (read or write) is denoted by r or w, respectively; each line segment is an operation, the start/finish point denotes the start/finish timestamp of the operation; for write operations, the value to be written is given in brackets; for read operations, the return value is in brackets. In this example, read operations r_1, r_2 begin after two write operations have finished. If this history is linearizable, both read operations should return the same value, which is the last updated value. However, r_1, r_2 return different values, indicating that this history is not linearizable.

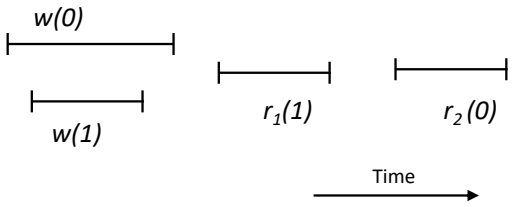


Figure 2.1: A non-linearizable history example.

Time-based Metric Γ

To quantify the staleness of operations in a history, Golab et al. [48] defined a time-based metric Γ. Γ quantifies the deviation from the given history to a linearizable history in time unit. In other words, it denotes the degree of changing needed by the means of

Γ-relaxation (defined later in this paragraph) to make the given history to be a linearizable one. The Γ-*relaxation* of a history is obtained by expanding each operation about its midpoint, specifically shifting the invocation and response times by $-\Gamma/2$ and $+\Gamma/2$ time units, respectively. The Γ-*value* of a history is the smallest Γ for which the Γ-relaxation of the history is linearizable.

For example, the non-linearizable history in Figure 2.1 has the Γ value shown in Figure 2.2. The Γ value equals to the duration from the finish time of $w(1)$ to the start time of $r_1(1)$. In other words, the history will be linearizable if we stretch the start time of and the finish time of each operation by $\Gamma/2$.

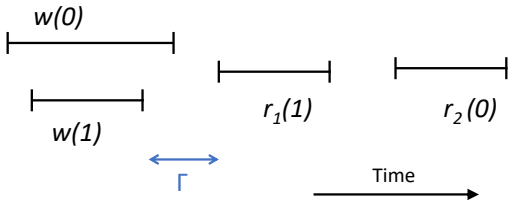


Figure 2.2: A non-linearizable history example shows Γ metric.

2.4 Summary

In this chapter, the concepts and theories related to this thesis were presented. Serializable isolation is a golden standard of strong transaction isolation: it is useful and easily understood because it provides the illusion of serial transaction execution. This chapter also discussed the transaction isolation and concurrency control mechanisms. In addition, distributed transactions require atomic commitment protocols to provide atomicity even in the case of failures. Following that, three distributed transaction protocols were presented: 2PL combined with 2PC and Calvin provide serializability; RAMP only guarantees a weak isolation – read atomicity. Eventually consistent systems, which provide availability even in the case of a network partition, trade consistency for availability and performance. Linearizability, a consistency standard for such systems, was covered. This chapter also presented the time-based Γ metric, which can be used to measure staleness of reads observed in eventually consistent systems.

Chapter 3

Epoch-based Concurrency Control and ALOHA-KV

3.1 Introduction

As motivated in Section 1.1, this chapter presents a distributed transaction protocol for read-only and write-only transactions as a counterexample against the long-standing assumption: concurrent serializable transactions under read-write or write-write conflicts require costly synchronization, and thus may incur a steep price in terms of performance [11, 12]. This protocol for high performance distributed transactions relies on the *epoch-based concurrency control* (ECC) mechanism, which is inspired by the slotted ALOHA network protocol [81].

ECC enables high parallelism for serializable read-only and write-only transactions. It does so by enforcing the execution of read-only and write-only transactions in alternating read-only and write-only epochs. Thus, read-write conflicts never occur in the execution. Within each epoch, transactions are processed in parallel with the help of several concurrency control techniques, such as multiversioning and timestamp ordering. The protocol requires very little metadata to be stored or to be transferred for each transaction, and yet achieves strong transaction isolation. The transaction execution in ECC uses a simplified atomic commitment protocol that only requires one round trip to commit a transaction in the absence of failures irrespective of contention, and uses a small number of additional messages whose cost is amortized across many transactions. ALOHA-KV, a scalable distributed key-value store for read-only write-only transactions, is built based on ECC and this transaction commitment protocol. ALOHA-KV can process close to 15 million read/write operations

per second per server when each transaction batches together thousands of such operations. The technical contributions presented in this chapter are as follows:

1. The chapter presents the epoch-based concurrency control for high performance distributed read-only and write-only transactions. ECC guarantees high parallelism in transaction processing even under high contention workloads that make many conventional concurrency control mechanisms suffer from blocking or retrying. The distributed transaction protocol and the theoretical analysis are presented in this chapter.

2. This chapter proposes a simplified atomic commitment protocol used in the ECC protocol. Different from 2PC, which requires two round trips for each write-only transaction even if no failure occurs, the proposed protocol only requires amortized one round trip.

3. The implementation and evaluation of a distributed key-value store with read-only and write-only transaction support, called ALOHA-KV, are presented. The experimental results show that when transaction size exceeds 10 key-value pairs, ALOHA-KV outperforms RAMP in terms of both throughput *and* latency, at the same time providing stronger transaction isolation.

The rest of this chapter is organized as follows. Section 3.2 introduces the architecture and system model of ALOHA-KV. Section 3.3 provides the design of ECC. The transaction protocol and implementation details are described in 3.4. Section 3.5 discusses the fault tolerance design. The theoretical analysis of the performance of the system is discussed in Section 3.6. Section 3.7 presents the performance evaluation of ALOHA-KV.

3.2 System Model and Architecture

ALOHA-KV is a scalable multi-version storage system that supports serializable read-only and write-only transactions across multiple data partitions. The protocol design targets throughput optimized systems. ALOHA-KV is therefore most suitable for applications that tolerate larger latencies and latency variations, although as shown in Section 3.7, it can be tuned to achieve a balance of latency and throughput that meets or exceeds a best-of-breed system.

Internally, ALOHA-KV uses a distributed transaction protocol that combines epoch-based concurrency control (ECC) for transaction isolation and an atomic commitment mechanism whose amortized complexity is one network round trip per transaction.

3.2.1 System Model

This section presents the system model, on which the design of ECC is based. For simplicity of presentation, replication and fault tolerance are not considered at this stage because they will be addressed in Section 3.5.

In memory DB. All data can reside in main memory, although persistent storage is available for logging and checkpointing as discussed in fault tolerance design in Section 3.5. The choice of in memory database is driven by several reasons: (1) hard disk/SSD performance is still orders of magnitude slower than main memory, while the storage class memory (SCM) has not been ready as a mainstream storage device [23]; (2) current commodity servers can easily install hundreds of GB to several TB of main memory per machine [1], and tens to hundreds of such servers are able to satisfy the storage capacity requirement of most enterprise applications [86]; (3) the demand of avoiding data distribution is less important because distributed transaction overhead is low in ALOHA-KV.

Horizontal partitioning. Each data item, identified by a key, has a single logical copy in its hash partition.

Read-only and write-only transactions. The system exploits the special structure of read-only and write-only transactions to minimize concurrency control overheads.

Multiversioning. Each write operation creates a new version of a data item, identified by a distinct version number. The system leverages object versions for concurrency control, but their main purpose is to support historical queries. The version number is therefore a timestamp generated by the system at the beginning of transaction processing. Versions older than a user-specified threshold can be removed by a garbage collector to free memory.

Timestamps and clocks. To generate a unique timestamp, a server combines its unique server ID, a monotonically increasing number, and the time from its local clock. For example, a unique timestamp uses the timestamp from the local clock for its most significant bits and the unique server ID for the following bits, and the monotonically increasing number takes the least significant bits. Tight clock synchronization across servers benefits performance

[1] The current generation of AWS EC2 x1e.8xlarge virtual machine instance is equipped with around 1 TB memory, an x1e.16xlarge instance has around 2 TB memory, and an x1e.32xlarge instance has around 4TB memory [10].

but is not required for correctness of ECC. Standard synchronization techniques suffice, such as NTP executed over a low-latency network. Current mainstream network infrastructure can easily provide round-trip time lower than hundreds of microseconds between two hosts within a single private cluster. Using a cluster in AWS EC2, which synchronizes the clocks to one of the hosts in the cluster, experiments presented in this chapter only observed dozens of microseconds (usually < 20 us) clock offset. This decentralized unique timestamp generation method may introduce a slight preference for writes performed by some servers (e.g., a server has a local clock slightly ahead of that of other servers). However, the slight preference can be counteracted by fairness policies, such as that clients should connect to servers using a round robin manner.

Serializable isolation. Committed transactions are serialized according to their timestamps. Write-only transactions are physically isolated from read-only transactions by the epoch-based concurrency control mechanism, which automatically deals with read-write conflicts. Conflicting write-only transactions are permitted to execute in parallel since they act on different versions of data. Read-only transactions that access old versions of data can be executed during a write epoch without causing conflicts as long as the version accessed precedes the start of the write epoch. Otherwise, such transactions must wait until the next read epoch.

3.2.2 Architecture

The architecture of ALOHA-KV, illustrated in Figure 3.1, comprises a collection of server backends (BEs), server frontends (FEs), and an epoch manager (EM).

FE: the transaction coordinator. An FE accepts transaction requests from clients, and acts as a transaction coordinator: it starts transaction execution during the correct epoch, generates a timestamp for each transaction, and communicates with the BEs to determine the outcome of each transaction. Clients may connect to any FE, and each FE may contact any BE where the required data items reside.

EM: decides epochs. The EM communicates with all FEs to control epoch changes, and thus determines when the FEs are able to start executing a given transaction. The durations of read and write epochs are either determined automatically by the EM, or tuned manually.

BE: the data store. A BE stores the data items in one partition of the database, and serves requests from FEs to read and write these items.

Deployment. The system is optimized for deployment in a private data center with a high-speed network. BE/FE pairs can be co-located on the same host, denoted by a

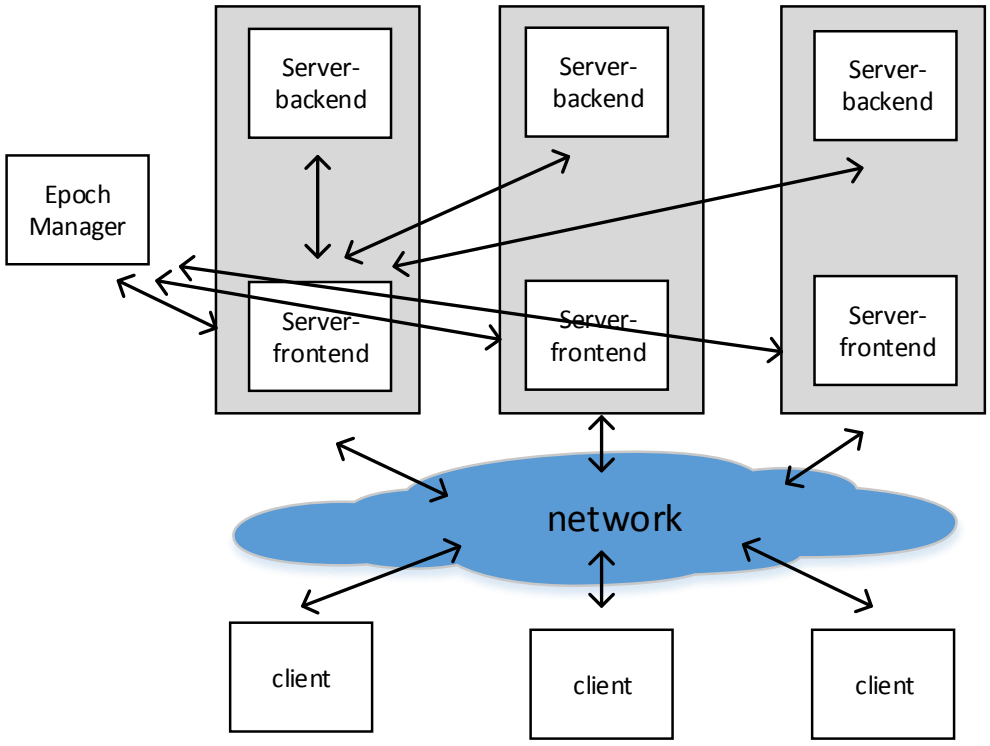


Figure 3.1: Illustration of the system architecture of ALOHA-KV.

gray box in Figure 3.1. Each system component can be replicated for fault tolerance, as discussed in Section 3.5. Furthermore, the epoch manager can be distributed for scalable performance, as discussed in Section 3.6.2.

3.3 Epoch-based Concurrency Control Mechanism

ALOHA-KV achieves transaction isolation using *Epoch-based Concurrency Control* (*ECC*). Conceptually, ECC is the combination of two techniques that maximize parallelism. First, ECC schedules read-only transactions and write-only transactions into disjoint time slots, called read-only and write-only epochs, to eliminate read-write conflicts. Note that ALOHA-KV accepts both types of transactions at all times, and merely delays the start of transaction

execution as needed to ensure that each transaction runs during the correct epoch type. For example, a write-only transaction accepted during a read epoch begins executing in the next write epoch. Second, ECC uses multiversioning to resolve write-write conflicts, which allows write-only transactions to proceed in parallel even when their write sets overlap. Both techniques combined ensure that transactions never abort or deadlock due to conflicts, which benefits throughput under contention.

In addition to dealing with conflicts efficiently, ECC minimizes communication overheads by simplifying atomic transaction commitment. Specifically, ECC cannot have "reads-from" dependencies among transactions executing within the same epoch, which means that the effects of a partially committed transaction cannot be observed until the next epoch. This enables one-phase commitment for write-only transactions, with a second phase required only if the transaction must be rolled back, for example due to an abort on failure. Any additional messages needed to orchestrate epoch switches are amortized over a large number of transactions. In contrast, two-phase commit requires both phases even when a transaction commits in the failure-free case. This section also describes the transaction barriers used to implement the epoch switching mechanism.

3.3.1 Invariants and Rules

The performance benefits of ECC are contingent on tight synchronization of the epoch status (read vs. write) across FEs. The synchronization mechanism records state information at the EM and FEs and guarantees a number of invariants with respect to this state.

Authorization. An FE can start processing a transaction only if it holds appropriate *authorization*, which is granted by the EM. An authorization comprises the epoch type (read or write), as well as two timestamps indicating a finite *validity period*. Transaction timestamps are always assigned within the validity period. An FE may hold at most one authorization at a time, ensuring exclusion among read-only and write-only transactions.

Epoch duration. From the point of view of an FE, an epoch is the period of time from when an authorization is granted by the EM to when the authorization is revoked, which is always after the end of the validity period.

Timestamp generation. A write-only transaction is assigned a timestamp when it is started by an FE. Recall that the timestamp is also the version number of the transaction. The FE guarantees that the timestamp is within the epoch's validity period to ensure that the serialization order of transactions, which is determined by the timestamps, is consistent with the order of epochs.

Transaction start policy. Write-only transactions can only be started under valid write authorization. Similarly, read-only transactions retrieving the latest data version can only be started under valid read authorization. However, read-only transactions accessing old versions of data (i.e., historical queries) can be started either under valid read authorization, or under valid write authorization if the version accessed precedes the start of the current validity period.

Transaction completion policy. A transaction that begins in one epoch must complete within the same epoch. Once the validity period of an epoch expires, an FE must wait for all pending transactions to complete before acknowledging to the EM that authorization has been revoked.

Total order of epochs. The EM must revoke all authorizations from FEs before granting a new authorization, which is tagged with a monotonically increasing ID. Epochs and their corresponding authorization validity periods are therefore disjoint.

Alternating epoch types. The EM drives the type of an epoch. In ALOHA-KV, the EM chooses an alternating sequence of read and write epochs, meaning that the epoch type changes each time an authorization is granted to the FEs. As discussed in Section 4.2.2, the EM has other options for deciding epoch types.

3.3.2 Transaction Barriers

For clarity of presentation, the pseudo-code presented later on in Section 3.4 omits the low-level details of the message passing protocol for epoch switching and instead uses a high-level API called *transaction barrier*. The API comprises two primitives called *Begin_Barrier* and *Finish_Barrier*. Begin_Barrier checks the FE's current epoch authorization and either admits the transaction if the authorization is valid and has the correct type, or else blocks the transaction until the correct authorization becomes valid. If the transaction is admitted, the count of in-flight transactions is increased by 1. Finish_Barrier retires transactions by updating the count of in-flight transactions, which must reach zero before the revocation of authorization.

3.3.3 Example

Figure 3.2 illustrates ECC using two FEs, each executing transactions in three worker threads. The epoch switch procedure works as follows:

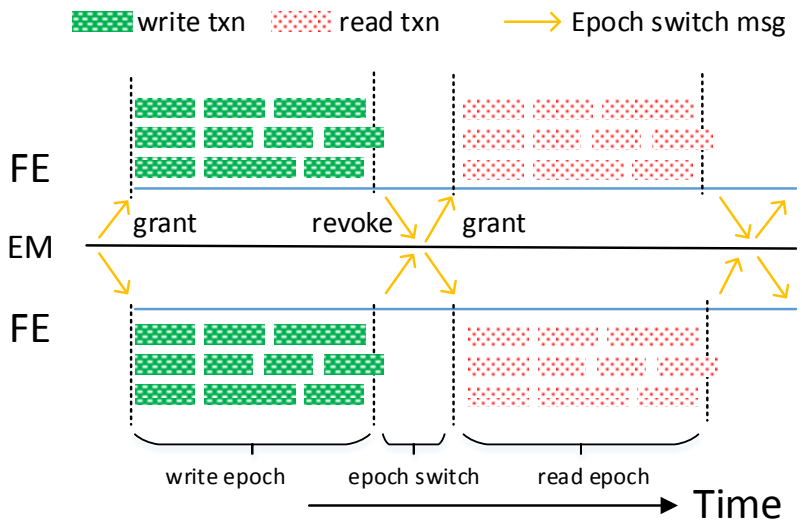


Figure 3.2: An example to illustrate the epochs in ECC.

1. The EM grants the FEs write authorization. The start and end of the validity period are indicated by vertical dashed lines in the figure. In practice, the validity period should be long enough so that the cost of epoch switching is amortized over many transactions.

2. In the write epoch, each worker thread executes write-only transactions until the validity period expires. The BEs simply store all the writes as new versions regardless of the order of the versions received, because these versions are not visible to reads during the write epoch. Thus, no write-only transaction should be blocked or aborted (unless it is aborted by transaction coordinator).

   In the meantime, read-only transactions accessing the latest version are buffered but historical queries can be admitted if they access sufficiently old versions.

3. Some pending transactions may continue to execute past the end of the validity period. All such transactions must be run to completion in the current write epoch before the next epoch begins.

4. Once all pending transactions are complete, the FE acknowledges to the EM that the write authorization is revoked. The FE now awaits a read authorization.

5. After the EM receives acknowledgments from all the FEs, it grants a read authorization for the next epoch.

3.4 Implementation

To evaluate the performance envelope of ECC, a key-value storage system is implemented, called ALOHA-KV, supporting read-only write-only transactions. This section discusses an atomic commitment protocol for read-only write-only transactions and some implementation details of ALOHA-KV. ALOHA-KV achieves high throughput thanks to efficiency in the transaction protocol and high parallelism enabled by ECC even under high contention.

3.4.1 Data Representation in Storage

ALOHA-KV stores key-value pairs in a hash-partitioned distributed table. The values are versioned to support historical queries, as well as to enable multiversion concurrency control for write-only transactions. In other words, each version of a value is represented as a pair of the form $\langle version, value \rangle$. For each key, the versions are organized in a logical list ordered by version, implemented as a linked list of arrays. Specifically, the data structure is composed of blocks, each of which includes a fixed size array of versions, and each block links to a block holding older versions. This data structure is a hybrid of linked list and array to accommodate removing old versions and efficiently accessing recent versions.

Inserting a new version of a key-value pair during a write-only transaction entails adding an entry to the array of the key, keeping the versions in sorted order. Since transactions with different timestamps execute in parallel during a write epoch, it is not always the case that the newly created version is the latest version. In other words, an older version may need to be inserted when the latest version has already been written by another transaction. Such an outdated version will not be visible to future read-only transactions that access the latest data, but can still be requested by a historical query. This is in contrast to conventional timestamp ordering with Thomas' write rule [21], where an outdated value need not be written at all. In practice, new versions tend to arrive in nearly sorted order since the version numbers are derived from timestamps, and this enables efficient insertion. Because all transaction timestamps are within the authorization validity period, there is no insertion of versions that predate the start timestamp of the write-only epoch.

Retrieving a value begins with determining the correct data version with respect to the transaction timestamp. This is either the latest version, if the transaction requests the latest data, or the latest version not exceeding a given version number, if the transaction is a historical query.

Deleting a version from the table occurs in two situations: when old versions are cleaned up by the garbage collector to recover memory, or when a transaction aborts in a write

epoch. In general, deleting a version entails removing the corresponding items from the array of the key. In the case of garbage collection, the deleted version must be older than a threshold that is no longer needed by any reads. Garbage collection can be triggered periodically, or when the system is low on memory.

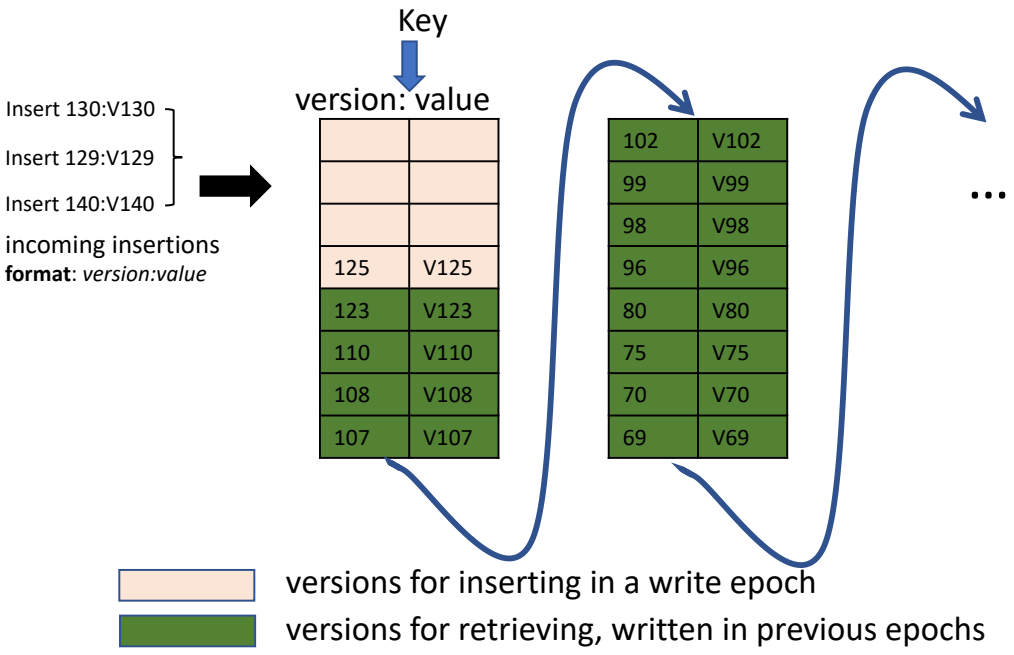


Figure 3.3: Illustration of multiversion storage and insertion in ALOHA-KV.

Figure 3.3 demonstrates an example of the key value store and some insertions. Insertions can be accepted in arbitrary order in a write epoch, but inserted versions are kept in order. The versions in green (darker) zone in the graph are historical versions written in previous write epochs. The version numbers to be inserted in write epochs must be within the validity period, so no version will be inserted to the green zone in this example (versions of previous epochs).

3.4.2 Transaction Protocol

The transaction protocol executed by the FE and BE is presented in Algorithms 1 and 2, respectively. The epoch switching mechanism and interaction with the EM is represented

implicitly in Algorithm 1 using transaction barriers, which were described earlier in Section 3.3. The entire protocol is implemented in C++ using fbthrift—a popular open-source RPC framework [36].

As presented in Algorithm 1, transaction execution begins with the invocation of the `PutAll`, `GetAllLatest`, or `GetAllHistorical` procedure at an FE, which acts as a coordinator. The FE executes a transaction barrier to ensure that it has appropriate authorization for the given transaction type, and then accesses the relevant keys in one or more partitions by invoking *partition requests*—calls to procedures `Put` and `Get` at BEs. `Abort` is called to roll back a write-only transaction.

For a write-only transaction, the FE first generates a unique timestamp ts (as described in Section 3.2) at line 3 of `PutAll`. It then builds a set of data versions at line 5, and distributes these versions to different partitions at line 7 by calling `Put` on different BEs. For simplicity, the pseudo-code shows separate calls to `Put` for each data item (similarly for `Abort` and `Get` later on), but in practice requests destined for the same BE are batched together. Each data version includes the key-value pair, transaction timestamp, epoch ID, as well as the transaction size (used in recovery, see Section 3.5.3). The `PutAll` procedure is regarded as successful if every call to `Put` has succeeded, otherwise if one or more calls fail then the FE invokes `Abort` on each partition involved to roll back the transaction. This is the second round of the atomic multi-write protocol.

A *read-only transaction* is executed similarly, but never needs a second round of messaging to abort. This is because a read-only transaction does not alter the state of a BE, and so there are no actions to roll back on failure. Procedure `GetAllLatest` returns the latest data versions for the given set of keys by calling procedure `Get` on respective partitions with a special timestamp value of $\perp$. Procedure `GetAllHistorical` executes a historical query and accepts a user-specified timestamp that defines the desired historical data version, which must not exceed a current timestamp that may be generated by the FE. Internally, the procedure behaves similarly to `GetAllLatest` but uses a transaction barrier only during a write epoch if the requested timestamp exceeds the start of the current validity period.

Procedures `Get`, `Put`, and `Abort` at BEs operate on the multiversion storage MV described in Section 3.4.1 for insertions, lookups, and aborts of key-value pairs. Procedure `Put` inserts versions and `Get` retrieves the latest version as of a given timestamp; `Abort` deletes entries and marks that version as aborted. The aborted mark prevents any future insertion for the same version, which may happen in the case that the `Abort` message is delivered before the `Put` message.

Algorithm 1: Transaction protocol for FE.

```
1  Procedure PutAll(W: set of ⟨key k, value v⟩)
2      Begin_Barrier_Put
3      ts = generate new timestamp
4      eid = current epoch ID
5      V = {(w.k, w.v, ts, eid, |W|) | w ∈ W}
6      parallel-for v ∈ V do
7          invoke Put(v) on respective partition
8      if any call to Put fails then
9          parallel-for v ∈ V do
10             invoke Abort(v) on respective partition
11     Finish_Barrier_Put
12 Procedure GetAllLatest(K: set of keys)
13     Begin_Barrier_Get
14     ts = generate new timestamp
15     Finish_Barrier_Get
16     parallel-for k ∈ K do
17         invoke Get(k, ts) on respective partition
18     return union of responses from calls to Get
19 Procedure GetAllHistorical(K: set of keys, ts: timestamp)
20     if holding write authorization with validity period starting at or before ts then
21         Begin_Barrier_Get
22         Finish_Barrier_Get
23     parallel-for k ∈ K do
24         invoke Get(k, ts) on respective partition
25     return union of responses from calls to Get
```

3.4.3 On the Side-Effects of Stragglers

A straggler transaction is one that prevents an FE from revoking authorization for a long time. It may delay the start of the next epoch for all FEs, and further degrade the overall throughput. Stragglers may arise from resource limitations during transaction processing, from long-running transactions, or from software/hardware anomalies.

In the absence of anomalies, long-delayed stragglers are unlikely to occur in ALOHA-KV for the following reasons. *First*, the number of in-flight transactions (preventing authorization revocation) decreases to zero after the epoch's finish timestamp is reached.

Algorithm 2: Transaction Protocol for BE.

Data: MV: multiversion storage described in Section 3.4.1
$txnsize$: transaction size of the write-only transaction for recovery (See Section 3.5.3).

```
1 Procedure Put(v: ⟨key, value, ts, eid, txnsize⟩)
2     return MV[key].insert(ts, v)
3 Procedure Abort(v: ⟨key, value, ts, eid, txnsize⟩)
4     MV[key].abort(ts)
5     return
6 Procedure Get(k: key, ts: timestamp)
7     return MV[k].get(ts)
```

As a result, in the course of an epoch switch, the contention among in-flight transactions tends to zero. *Second*, the ALOHA-KV system only handles one-shot transactions and is able to process them quickly by reading and writing in-memory data within epochs. In ALOHA-KV experiments presented in this chapter, long-running stragglers that stall the system by more than one epoch duration were not observed even for transactions including thousands of keys.

Another observation is that slow read-only transactions will not block revoking authorization and delay starting of the next write epoch. This is because the read-only transaction only accesses historical versions after it gets a timestamp at line 14 of Algorithm 1 and then releases the barrier at line 15.

3.5 Fault Tolerance

ALOHA-KV relies on main memory storage for performance, and therefore depends crucially on appropriate fault tolerance mechanisms to protect against data loss and maintain system availability in the event of a server failure. This subsection explains how the strategies of replication, logging, and checkpointing to persistent storage are applied to different components of the system.

3.5.1 Replication

BE replication. Replication is essential for BE servers, which store key-value pairs. During write epochs, the FE writes primary BEs as well as backup BEs. When the primary fails,

the backup can replace the failed primary server with a configuration update as described in Section 3.5.2. Note that in ECC, write-only transactions can achieve both concurrency control and replication in one round trip amortized if there are no aborts. In comparison, traditional 2PL/2PC with primary-backup requires two rounds for 2PC plus an extra round for replication. During read epochs, load balancing is possible between primary and backup servers because they hold exactly the same data.

Fast epoch switch at FE/EM failure. Failures of FE servers do not lead to loss of data, but may stall the ECC mechanism. This is because the EM must receive acknowledgments of authorization revocation from the previous epoch before granting authorization for the next epoch. The backup FE is therefore charged with completing all pending transactions and reporting back to the EM if the primary FE fails during a write epoch. This requires that the primary forwards each transaction to the backup at the beginning of transaction execution, and confirms to the backup after execution. If the primary FE fails, the backup acts temporarily as the coordinator, simply aborting all in-flight transactions in the interest of a fast epoch switch. Once a BE is contacted by the backup FE, it deems the primary is failed and rejects any further requests from the primary FE within the epoch. This mechanism is similar to [16, 46] and deals with the anomaly where both primary and backup believe they are the primary. As clients continue to send requests to FEs, the failed FE is not allowed to participate in future epochs because it does not have the latest cluster membership configuration (see Section 3.5.2). In the course of an epoch switch the FEs become nearly idle and so aggressive failure detection (e.g., based on a sub-second heartbeat) can be used.

Similarly, failure of the EM can lead to loss of synchronization among FEs, for example with some servers starting a new epoch while others await authorization. When the primary EM fails, the backup first polls the FEs to determine how many of them have not yet acknowledged authorization revocation for the most recent epoch. When the outstanding acknowledgments are received, the backup EM may resume ordinary execution by granting the next authorization. EMs use a handover mechanism and aggressive failure detection similarly to FEs.

3.5.2 Cluster Membership

Each server in ALOHA-KV relies on a configuration of cluster membership to discover and to name other servers within the system. The membership design of ALOHA-KV is similar to the membership configuration in Corfu [16, 46].

The configuration includes the information of all active servers, such as addresses and

roles (e.g., BE, BE backup). The configuration is versioned with an increasing version number. A server can easily detect a peer server in communication that has an obsolete version of configuration, because each sever of ALOHA-KV must tag requests with the version number in communication messages. Thus, the anomaly that both the primary and the backup believe they are the primary will never happen, because only the one having latest configuration will be recognized by other servers. When a server which previously had active status in the configuration fails, a configuration update is needed within the cluster, which is usually initiated by the server which first detects the failure. The new configuration contains all the active servers in the perspective of the proposer, and it must be accepted by all the active servers in the new configuration via two-phase commit. To avoid the "split-brain problem," that is two set of servers maintain two separate memberships, the active servers in the new configuration must be the majority of the initial membership setting.

3.5.3 Logging and Checkpointing

To protect data against a system-wide failure, such as loss of power to an entire rack of servers or data center, ALOHA-KV persists data by logging operations to secondary storage. Specifically, the BEs log all the requests they receive from FEs during write epochs. To avoid a performance bottleneck, this design opts for an asynchronous form of logging whereby worker threads append log entries to an in-memory buffer that is flushed periodically by a dedicated thread. It is ensured that all data are flushed to disk before the next epoch starts, thus enabling the following guarantees: (1) a transaction executed in a failure-free write epoch is never lost; (2) a write-only transaction whose effects are observed by a read-only transaction is never lost; and (3) transaction recovery is atomic: either all or none of the operations in a transaction are recovered.

A checkpoint is a persistent snapshot of the whole database. As a multiversion system, ALOHA-KV can create a consistent checkpoint simply by dumping the versions of all keys before a given timestamp. This design creates checkpoints only at an epoch finish timestamp for easy recovery.

Recovery logs are organized into a collection of files, with one file per BE per write epoch. Each log file is capped with a special record at the end of a write epoch to indicate that it is complete. After a failure, the recovery procedure on each partition first reloads a checkpoint and all completed log files. For atomicity of recovering committed transactions in the uncompleted logs, logs from the most recent epoch are then sent to a recovery coordinator, which checks for each transaction whether the number of operations present in the logs

matches the size of the write set, which is recorded in the partition requests (see line 5 in Algorithm 1). For atomicity, any transactions having a full complement of operations are replayed, and the remaining transactions found in incomplete logs are aborted.

3.6 Theoretical Analysis

The performance envelope of ALOHA-KV is substantially different from other systems because ECC schedules transactions into epochs. This section discusses the factors affecting the latency and throughput of the system. This section defines throughput as the number of key-value pair operations per second, which in the experiments equals the number of transactions executed per second times the (fixed) transaction size. This section also sketches out a proof of serializability for ECC.

3.6.1 System Throughput

The ECC protocol in ALOHA-KV periodically switches between write epochs and read epochs. Consider a unit of execution containing one write epoch followed by one read epoch. Let P denote the overall throughput in the unit, P_W the throughput of writes in the unit, P_w the throughput of writes in the write epoch, P_R and P_r denote analogous quantities for reads. Let t_w, t_s, t_r denote the duration of a write epoch, the epoch switch time, and the duration of a read epoch, respectively. Let n_w denote the number of write operations executed in the write epoch. The overall write throughput is

$$P_W = \frac{n_w}{t_w + 2t_s + t_r} \tag{3.1}$$

which can be rewritten as

$$P_W = P_w \times \frac{t_w}{(t_w + 2t_s + t_r)} \tag{3.2}$$

since $P_w = n_w/t_w$. Using a similar equation for P_r, total throughput can be expressed as follows:

$$P = P_w \times \frac{t_w}{(t_w + 2t_s + t_r)} + P_r \times \frac{t_r}{(t_w + 2t_s + t_r)} \tag{3.3}$$

Equation 3.3 suggests the following strategies to maximize throughput: increase P_w and P_r, decrease the epoch switch time t_s, and increase the epoch durations t_w and t_r.

For the first strategy, the main factors affecting P_w and P_r are as follows:

- *Transaction size.* Larger transactions benefit from better network and processing efficiency due to batching, but they are more likely to overrun the validity period of an epoch, which complicates epoch switching.
- *Number of servers.* On the one hand, adding servers increases I/O and processing capacity. On the other hand, it increases the number of partitions and hence causes transactions to be processed by BEs in smaller pieces, which counters the benefits of batch processing.

For the second strategy, one can observe that epoch switch time t_s roughly amounts to the sum of one round of communication between the EM and FEs, and the time required to complete any pending transactions. In particular, t_s is highly dependent on the slowest FE to acknowledge authorization revocation. The following strategies are proposed to reduce t_s:

- *Minimize EM-FE communication latency.* For example, use a separate execution path for epoch switch messages versus transaction messages, such as using a dedicated and prioritized thread for epoch switch.
- *Handle stragglers individually.* Some servers may be consistently slower than others due to differences in hardware configuration or network connectivity, in which case they can be authorized by the EM for shorter epoch durations.

On clock offset. The clock offset between any FE and the EM cannot violate the serializability guarantee of ECC, because an FE must assign a transaction timestamp within the validity period given by the EM. In other words, the timestamp lies in the intersection of the time after when an FE receives the authorization and the FE's interpretations of the validity period according to its local clock. Thus, if an FE has a large clock offset relative to the EM, the period during which an FE can generate timestamps and process transactions may be small, but other FEs are not affected. Moreover, the clock offset among servers of a private cluster (one of them is the time source) in the AWS EC2 environment using NTP protocol, is usually dozens of microseconds, which is far less than 1% of the epoch duration used in the experiments in this chapter.

3.6.2 Epoch Switch Scalability

The transaction barrier is one of the main factors affecting system performance, as explained in Section 3.6.1, and therefore two design alternatives are considered: a centralized approach

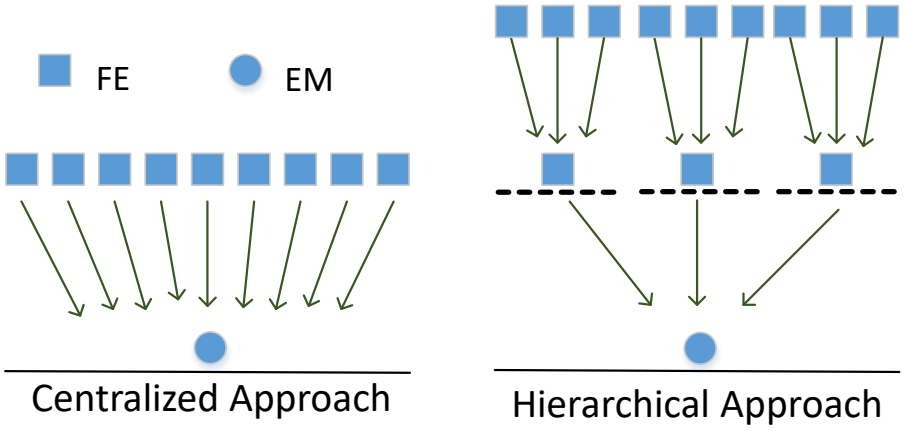


Figure 3.4: Barrier synchronization.

and a tree-structured hierarchical approach. This subsection will develop performance models for both alternatives.

Two criteria for scalability are identified in [14] as *synchronization independence* and *partition independence*. Informally, synchronization independence means one client's transaction will not cause another client's transaction to stall or fail. Partition independence means clients only need to contact partitions that store the data items in the transaction's read and write sets. Partition independence holds in the ECC protocol. Transactions within one epoch will not stall or fail each other, and a distributed barrier synchronization pattern is used to synchronize transactions from different epochs. Specifically, each FE reaches the barrier when it finishes all transactions in the epoch and is revoked authorization by the EM. After all FEs reach the barrier, the system will proceed to the next step (next epoch).

Barrier synchronization [51, 57, 99] is a synchronization mechanism used in hardware and software parallel computing, such as shared memory multiprocessor parallel processing or MPI (message passing interface) synchronization. A barrier mechanism allows massively parallel processing systems to synchronize all processing elements in a short time. For example, the prevalent Pregel [68] paradigm uses BSP (bulk synchronous parallel) model [96], which splits processing into computation, communication and barrier synchronization phases.

Considering how to synchronize epoch status scalably, Figure 3.4 shows two patterns of organization. The *centralized approach* allows messages to pass concurrently, but the message processing is done by one host. The ECC protocol presented in the previous section belongs to this pattern. In this pattern, all FEs are directly contacted by the EM for barrier synchronization. The *tree-structured hierarchical approach* in the figure presents

a tree structure with degree d equal to 3, in which messages are passed by the paths from leaves to root node, but each node only processes d messages. This pattern is similar to combining tree barrier [51, 99], aiming to handle a larger scale system. In this pattern, the EM is the root of the tree, the non-leaf non-root node can be either an FE or a special entity combining and forwarding synchronization messages from the child nodes to the parent node or vice versa. The remainder of this section will formally compare the two patterns for epoch switching. Based on Section 3.6.1, it is desirable to minimize epoch switch time for better performance.

In the centralized approach, the epoch switch phase entails both message passing over the network and message processing to count how many FEs have reached the barrier. For simplicity, it assumes the round-trip network latency between hosts is a constant L_n, and that processing a message from one host also takes a constant time L_p. Assuming that all FEs finish pending transactions at the same time, the epoch switch latency L of the barrier can be written as

$$L = L_n + L_p \times n \tag{3.4}$$

where n denotes the number of FEs.

In the hierarchical approach, let d denote the degree of the tree. In this model, the tree height is $\lceil \log_d n \rceil$, denoting the number of network hops from an FE to the EM in the tree, and each non-leaf node is expected to process d messages from child nodes. Thus, L in this model is

$$L = (L_n + d \times L_p) \times \lceil \log_d n \rceil \tag{3.5}$$

Note that equation 3.4 is a special case of 3.5 when $d = n$.

Finally, if FEs finish pending transaction processing at different times, and the slowest FE takes L_f time to finish pending transactions, the latency can be bounded as

$$L \leq L_f + (L_n + d \times L_p) \times \lceil \log_d n \rceil \tag{3.6}$$

which is tight when the slowest FE is a leaf node in the tree topology.

Formula 3.6 suggests that when network latency is low and message processing is expensive, a smaller d and larger tree height yield lower epoch switch latency L. Otherwise a larger d is preferred, which in the extreme case $d = n$ makes the hierarchical and centralized alternatives equivalent. Asymptotically, a hierarchical barrier has better complexity in terms of n ($O(\log n)$ in formula 3.5) than centralized ($O(n)$ in formula 3.4), but experimentally (see Section 3.7.4) the centralized approach yields very fast epoch switch times even with hundreds of FE nodes.

3.6.3 Analysis of Serializability

Informally, serializability provides the illusion that transactions are processed in a serial order [21, 73]. In particular, a history is one-copy serializable (1SR) if it is equivalent to a one-copy serial MV history, by definition presented in Section 2.1.3. The following sketches a proof that ALOHA-KV provides serializability.

Lemma 1. *Every history of committed transactions generated by ECC is one-copy serializable (1SR).*

Proof sketch. Given a history H of committed transactions, assign a timestamp to each transaction as follows: for a write-only or read-only transaction use the timestamp assigned by the FE at line 3 and at line 14 of Algorithm 1, respectively; for a historical read-only transaction use the timestamp passed by the client to the `GetAllHistorical` procedure. Now arrange the transactions into a serial history S in increasing order of their timestamp, breaking ties arbitrarily for read-only transactions. It follows easily that H and S have the same committed transactions and operations.

This paragraph presents the proof that H and S have the same "reads-from" relationships. In H, each read-only transaction obtains the highest version of a key that does not exceed its own timestamp, and this version is created by a unique write-only transaction. The proof is by contradiction. Assume some read-only transaction T does not obtain the highest version ts' of a key that does not exceed its own timestamp ts, where $ts' < ts$. Then this transaction T is executed before the transaction T' that creates the version ts'. Let E' and E denote the epochs in which the write-only transaction T' and the read-only transaction T are executed. An epoch *precedes* another epoch when its authorization period finish timestamp is before the authorization period start timestamp of the other epoch. If E' precedes E then T would have seen T', contrary to the earlier assumption, and so E precedes E'. The authorization periods for E and E' are disjoint no matter how badly skewed the local clocks of servers are, because the EM enforces this when it assigns the authorization period timestamps. Thus, E preceding E' implies that T has a lower timestamp than T' because ts' must be within the authorization period of E' and ts must be smaller than the authorization period finish timestamp of T. This implies that $ts < ts'$, which contradicts with the earlier observation that $ts' < ts$.

As a result, H and S have the same "reads-from" relationships, and moreover S is one-copy serializable because S is a serial history. This implies that H is 1SR. □

3.7 Experimental Evaluation

The experiments presented in this section demonstrate that ALOHA-KV performs favorably compared to RAMP [14] in terms of both throughput and latency for large transaction size (>6 for RAMP-S, >10 for RAMP-F). Micro-benchmark results show that although the latency of transactions in ALOHA-KV grows linearly with the epoch duration, short epochs (tens of ms) are sufficient to attain throughput levels close to the limit of the performance envelope. ALOHA-KV is able to process around 230 million operations per second for large transactions using fifteen servers. This section also shows that a single EM can orchestrate epoch switches in a timely manner even when controlling hundreds of FEs.

3.7.1 Experimental Setup

The experiments were deployed in Amazon EC2, using c3.8xlarge virtual machine instances in a single availability zone. Clocks across hosts were synchronized using NTP to a double-digit microsecond clock offset (usually $< 20\mu s$).

The experiments use the following default settings: five client hosts and five server hosts running a co-located BE/FE pair (different processes), with one server host running the EM. Data items comprise eight-byte keys and values. This experiment varies the transaction size, defined as the number of key/value pairs accessed per transaction. Keys are drawn uniformly at random from a space of 1 million elements. The ratio of write-only to read-only transactions is 1:1 to target both read-intensive and write-intensive workloads. ALOHA-KV uses alternating read epochs and write epochs with an epoch duration of 20 ms. Replication and logging (see Section 3.5) are disabled by default.

For comparison, a *baseline* system that represents an upper bound for performance with respect to the chosen implementation language and RPC framework is implemented. In the baseline, FEs and BEs process transactions without any concurrency control or atomic commitment protocol, and there is no epoch switching. It also executes RAMP on the same infrastructure with the same workload settings. Experimental measurements focus on average latency and aggregate throughput of operations (transaction throughput times transaction size), because it is more interesting to see the performance trade-offs of batching operations into transactions. However, this section presents both the throughput of operations and of transactions, both of which are used in the RAMP paper [14]. Each data point represents the average of three runs, and is plotted with error bars indicating the min and max measurements. In many cases, *the error bars are imperceptibly small.*

3.7.2 ALOHA-KV vs. RAMP Results

First, this section compares ALOHA-KV with the recently published transaction protocol RAMP, which supports weaker-than-serializable distributed read-only and write-only transactions. This section does not include a 2PC/2PL baseline in the experiments because RAMP exhibits significant performance gains over these techniques [14]. Published RAMP experiments consider only small transaction size (default 4, maximum 128), whereas it is also interesting to explore the performance of larger transactions, because smaller transactions can be combined into larger ones for efficiency. For a fair comparison, the experiment uses 5000 synchronous RPC clients in total (same as [14]). Of the three protocol variations in [14], this section tested RAMP-Fast and RAMP-Small only since the performance of RAMP-Hybrid is known to be a compromise between the other two.

Figure 3.5 shows that ALOHA-KV outperforms RAMP in terms of both throughput and latency for large transactions (around >6 for RAMP-S, >10 for RAMP-F). The performance gaps grow as transaction size increases. For transaction size 1000, the system achieves throughput roughly 20× greater than RAMP-S, and over three orders of magnitude greater than RAMP-F. ALOHA-KV scales almost linearly with transaction size in terms of throughput, which demonstrates the benefits of batching operations. It is observed that ALOHA-KV performance follows closely the baseline implementation despite providing atomic transaction commitment and serializable isolation.

As reported in the RAMP paper, RAMP-F performs worse than RAMP-S for larger transactions, with the performance gap widening as transaction size increases. This is because RAMP-F metadata stored in the storage system grows linearly with transaction size for each key, and large metadata impact performance for both reads and writes. In the worst case, a read-only transaction of size n may retrieve metadata referring to n^2 keys if each key in the transaction returns metadata containing n other keys. In comparison, the latency of RAMP-S under various transaction sizes is much flatter because it uses constant size metadata stored in the storage system. However, RAMP-S requires two round trips for all read transactions and transmits metadata whose size is linear in the transaction size in the second round of messaging, which explains the lower throughput relative to ALOHA-KV.

For smaller transaction sizes, RAMP outperforms ALOHA-KV slightly, because the metadata overhead in RAMP is small and because ALOHA-KV incurs overhead for epoch switching. Furthermore, the RPC framework differences also account for performance differences in small transaction size. For example at transaction size 1, the throughput of RAMP-F and RAMP-S are 31% and 9.2% higher, respectively, than the baseline (no concurrency control, no atomic commitment). However, the throughput of ALOHA-KV and

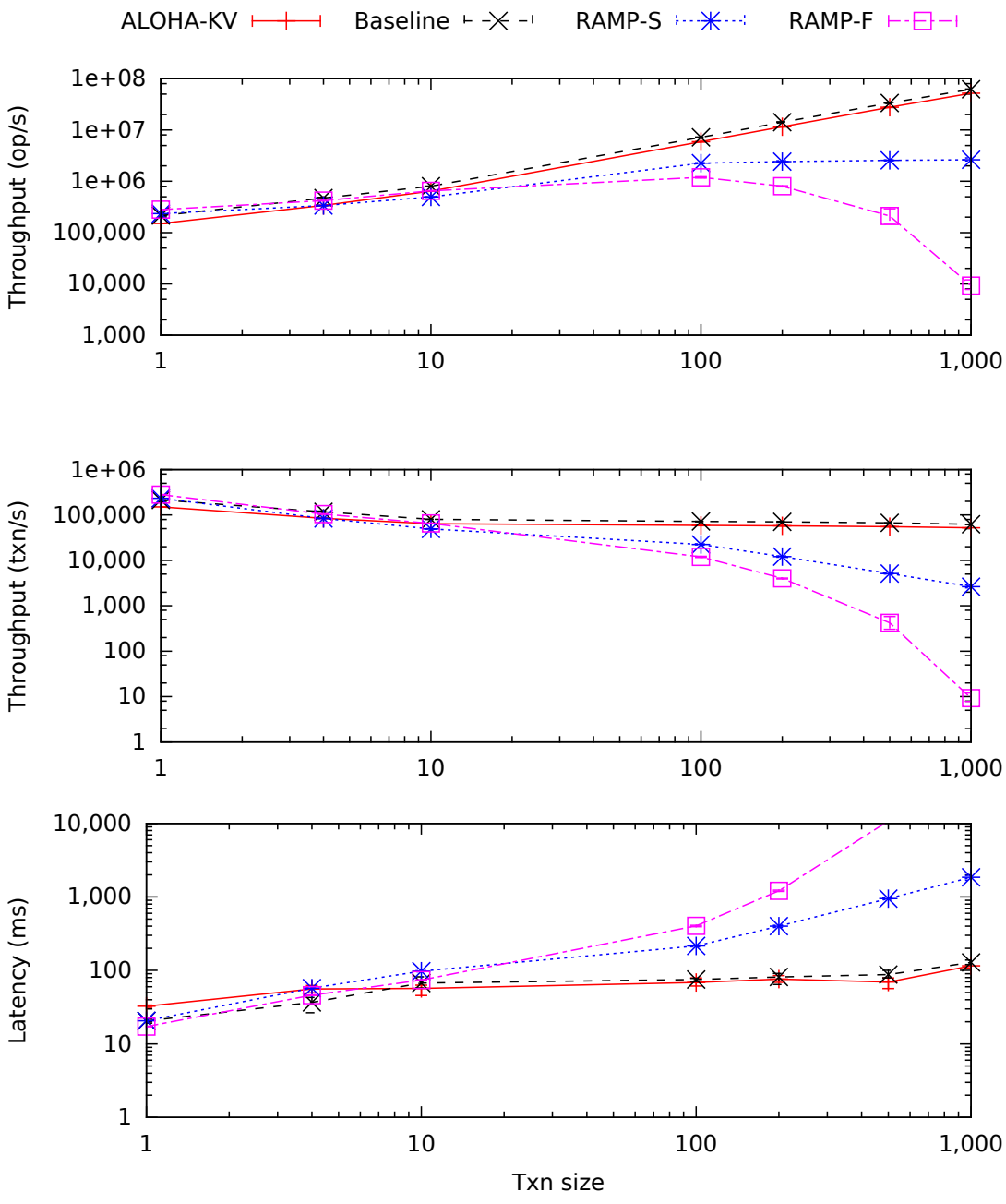


Figure 3.5: Experiment results of throughput and latency: ALOHA-KV vs. RAMP under 20ms read/write epoch duration. Throughput and latency presented using logarithmic scales.

RAMP-S for large transactions can indicate the performance of batch writing of small-size transactions, because metadata size is irrelevant to transaction size in these algorithms.

In summary, even though the RAMP protocol provides both synchronization and partition independence [14], the potential benefits of larger transactions are counteracted by the overheads of large metadata in RAMP-F and large messages in RAMP-S. In contrast, ALOHA-KV records minimal metadata stored in the storage system and transmitted in the protocol messages.

3.7.3 Microbenchmark Experiments

To better understand the performance of ALOHA-KV under various scenarios, this section presents the results of microbenchmark experiments. Unless otherwise specified, these experiments use the same deployment as in Section 3.7.1. Clients use the fbthrift async-client API, which allows issuing multiple requests without blocking on the response, and enables potentially higher throughput than synchronous RPCs. The experiments use transaction size 1000, 50% read-only transactions, and 100ms read and write epoch duration as default settings, to simulate workloads of a batch processing system [26, 54] (i.e., large transaction size, balanced reads and writes).

Epoch duration

Figure 3.6 shows results under various epoch durations. The experiments use 80 async clients, which ensures that the ALOHA servers are not overloaded. The results confirm the intuition that longer epochs yield both higher throughput by reducing the proportion of time spent on epoch switching, and higher latency by making transactions wait longer for authorization. Latency grows nearly linearly with epoch duration, whereas throughput is fairly flat beyond about 50ms.

The experiment also shows that epoch switching has relatively little impact on performance. In particular, there is only around 10% throughput difference between epoch durations of 10ms and 100ms. Thus, most of the throughput benefits of ECC are realized with fairly short epochs, and little penalty in terms of latency. It is noticed that under large epoch duration (≥ 50), average latencies are less than half of epoch duration. This is because transactions arriving during the correct epoch enjoy a low latency, and transactions that arrive out-of-epoch must wait half of an epoch duration on average before they can begin executing. The average of the two cases is less than half of the epoch duration when the epoch duration is large.

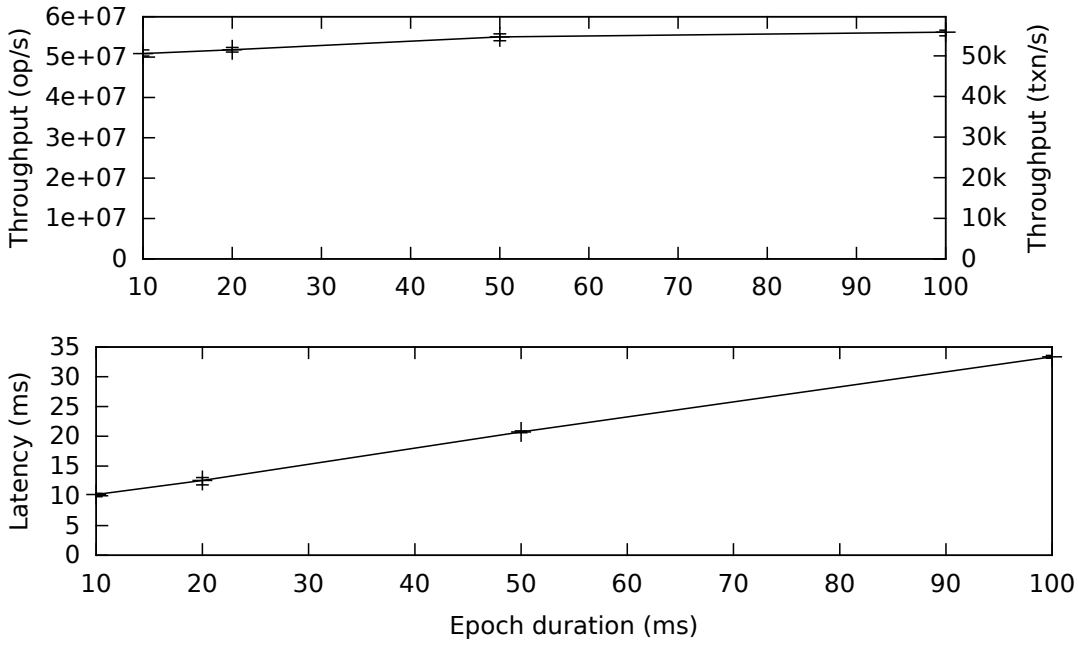


Figure 3.6: Throughput and average latency experiment results of ALOHA-KV under various epoch durations.

Transaction size

Figure 3.7 illustrates the effect of transaction size on throughput, which increases rapidly up to roughly 1000 operations per transaction. This is because batching boosts network and processing efficiency. Beyond 2000 operations per transaction, the system exhibits diminishing returns, and plateaus around 100 Mops/s. Since the transaction coordinator chops transactions into fragments sent to different partitions, it is expected that a larger transaction size is needed to saturate the throughput as the number of servers increases.

Proportion of read-only transactions

In experiments pertaining to the proportion of read-only vs. write-only transactions, the two cases are considered: when the proportion is known, and when the proportion is unknown and the default epoch durations are used. These cases are denoted as *adaptive epochs* and *fixed epochs* in Figure 3.8. This section also presents the performance of the baseline, where transactions execute without concurrency control and no atomic commitment protocol is

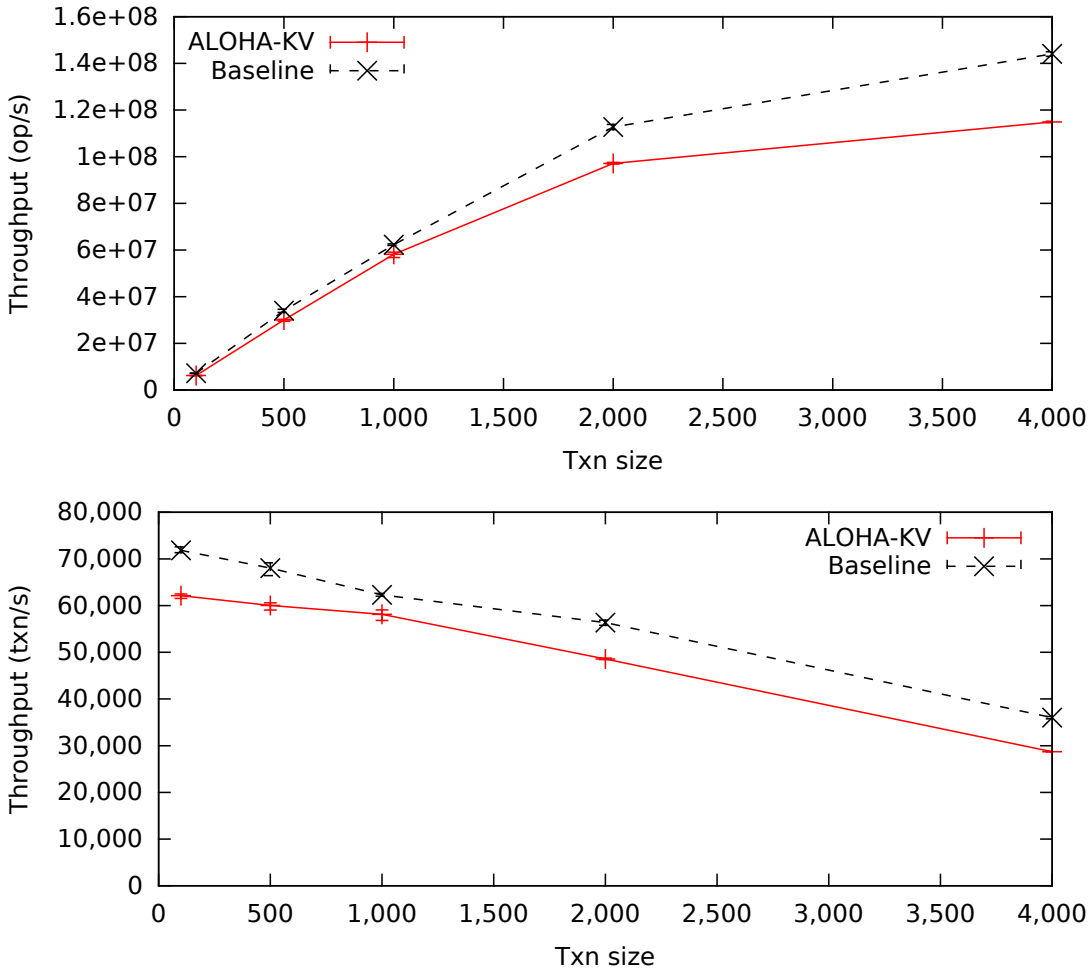


Figure 3.7: Throughput experiment results of ALOHA-KV under various transaction sizes.

used.

The results show that even without any prior knowledge of the read proportion, throughput using fixed epochs is roughly half or more of the level observed using adaptive epochs. The results for adaptive epochs are representative of cases when the workload exhibits a steady read/write mixture that is either known a priori, can be predicted accurately, or can be measured on-the-fly. The results also demonstrate low overhead for serializable transactions in ALOHA-KV under various read ratios as compared with the baseline.

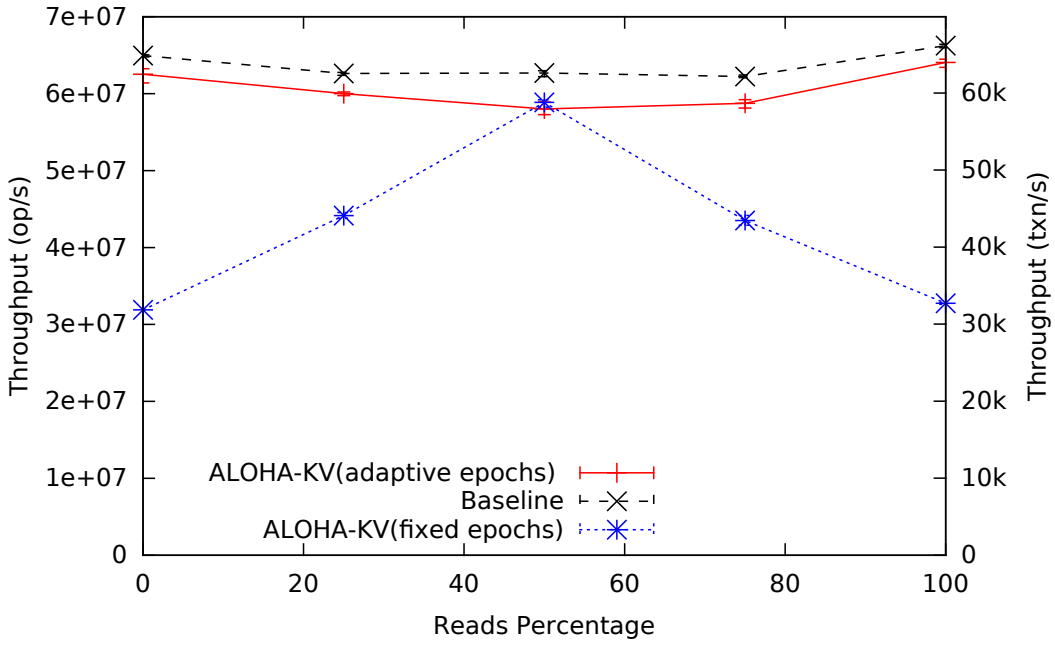


Figure 3.8: Throughput under various experiment results of ALOHA-KV read/write proportions.

3.7.4 Scalability

The experiments evaluated the scalability of ALOHA-KV with respect to different numbers of servers from two angles: (1) the scale-out throughput performance, (2) the overhead of epoch switching. In the first case, the experiments run ALOHA-KV up to a throughput of 233 million ops/s, achieving close to linear throughput scalability. In the second case, the result shows that a single EM can control hundreds of FEs with only single-digit-millisecond overhead per epoch switch.

Scale-out

The experiments vary the number of servers hosting BE/FE pairs. The transaction size is set to 4000 to allow batching a significant number of operations in messages sent to each BE. Figure 3.9 shows that using up to 15 BE/FE servers, ALOHA-KV achieves around 233 Mops/s. Since the probability of conflicts among write-only transactions grows with the observed throughput, the results demonstrate that ECC sustains a high degree

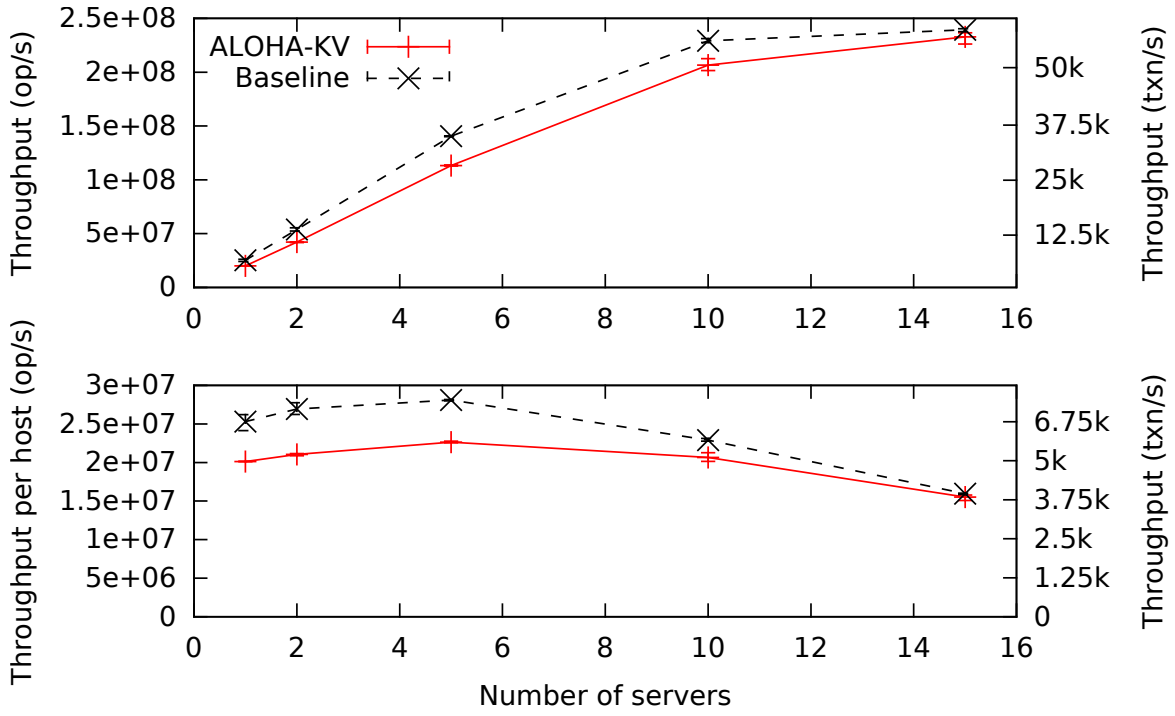


Figure 3.9: Aggregate and per host throughput experiment results of ALOHA-KV using transaction size 4000.

of parallelism despite contention, because writes are stored to different versions using multiversioning. Throughput per server drops slightly as the number of servers increases due to less effective batching, leading to sub-linear scalability. In the extreme case of 1 or 2 servers, the per-server performance is slightly lower than that of 5 servers, due to the single host network performance limit. However, when more servers are added to the system, additional network resources are utilized by the servers.

Epoch switching overhead

As discussed in Section 3.6.2, epoch management can be implemented either using a single EM, or by organizing the FEs in a scalable hierarchical structure. This experiment investigates whether a single EM has enough processing capacity to control hundreds of FEs centrally. To isolate the epoch switch time, the epoch duration is set to zero, meaning that each FE responds immediately to an authorization grant by acknowledging authorization

revocation. Upon receiving responses from all FEs, the EM issues the next round of epoch switch requests. Each FE instance is run on a distinct physical core, and there is no client workload.

Figure 3.10 shows the average epoch switch time for up to 640 FEs. The epoch switch time grows nearly linearly with the number of FEs, reaching 3.4ms at 640 FEs. As the number of FEs increases, the EM needs more time to process epoch switch messages, whose number grows linearly with the number of FEs. When the FEs are under load from clients, it is expected that epoch switch times will be longer than in this experiment because each FE must finish any in-flight transactions before responding to the EM at the end of an epoch. However, in that case the EM will be more lightly loaded because it has more time to process the same number of messages. Based on the experimental data, it is concluded that a centralized EM is sufficient for controlling a cluster of hundreds of ALOHA-KV servers.

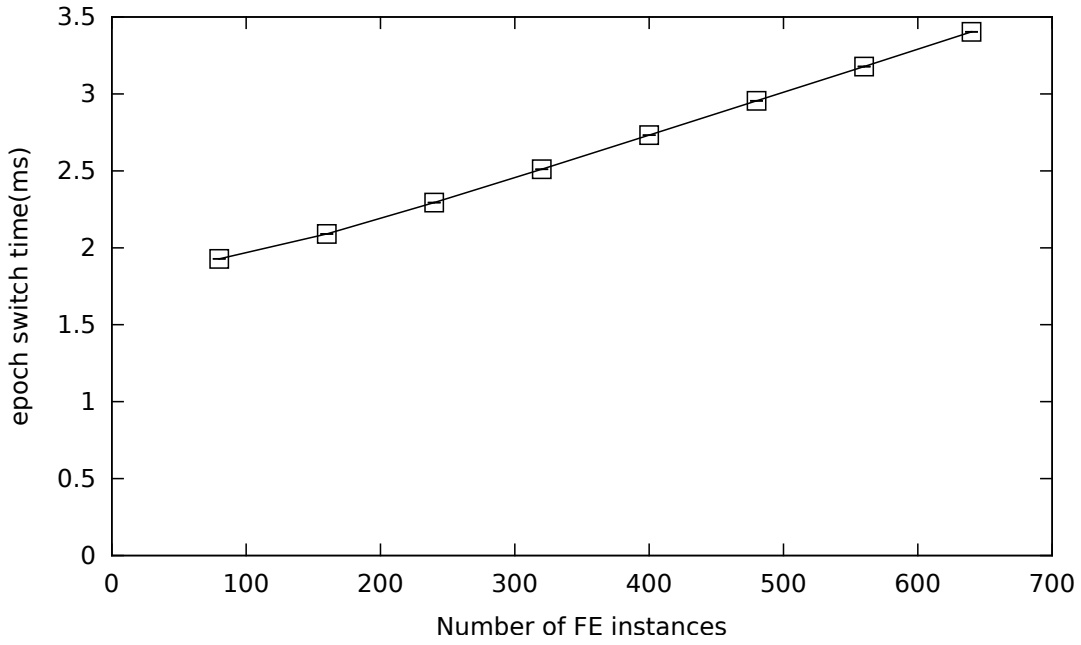


Figure 3.10: Epoch switch time for various numbers of FE instances in experiments of ALOHA-KV.

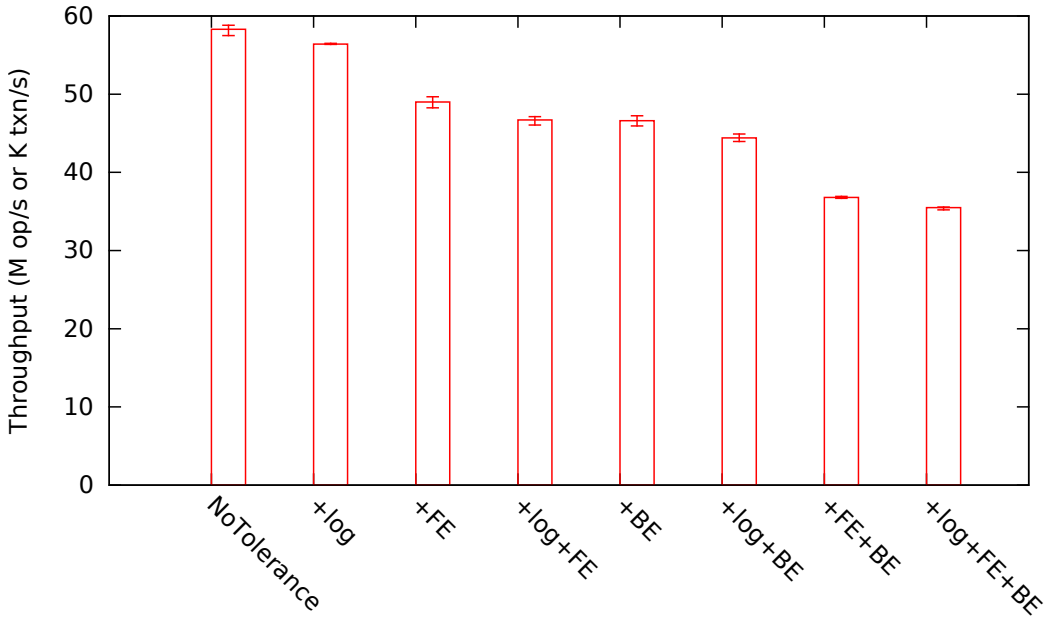


Figure 3.11: Evaluation result of various fault tolerance strategies in experiments of ALOHA-KV.

3.7.5 Fault Tolerance

Figure 3.11 shows the performance of various fault tolerance strategies. The system has five partitions, and each partition has a primary server and a backup server in the experiments. The SSD disks in the virtual machines are used for logging. In the table, "+log" denotes turning on the logging strategy of BEs writing operation logs to disk; "+FE" denotes turning on the FE replication that use backup FEs to take over the coordination when the primary crashes; and "+BE" denotes turning on BE replication that uses backup BEs to store a copy of the data held by the primary.

The results show that "+log", "+FE", and "+BE" lead to throughput penalties of 2–5%, 16–21%, and 20–25%, respectively. The asynchronously logging strategy "+log" (described in Section 3.5.3) achieves low overhead because the cost of flushing at the end of each epoch is hidden in the epoch switch costs. The "+log+FE" strategy has low overhead and protects against data loss even if an FE or BE crashes (see Section 3.5). The most expensive strategy is "+log+FE+BE", which achieves 35.5 Mops/s—roughly 40% slower than no fault tolerance. In comparison, RAMP performance in some cases is substantially lower without any replication.

3.8 Summary

This chapter addressed the problem of supporting high-throughput multi-partition read-only and write-only transactions, a.k.a *multi-put* and *multi-get*. This chapter proposed the ECC mechanism for these transactions, which guarantees serializability. ECC avoids read-write conflicts among transactions by partitioning transaction execution into disjoint read and write epochs, and mitigates write-write conflicts by storing multiple versions of key-value pairs. Thus, concurrent writes can be processed in parallel with low overhead, even when their write sets overlap. Using ECC as the central building block, this chapter described a distributed protocol for serializable read-only and write-only transactions, which requires amortized one round trip to commit a transaction in the absence of failures even under update intensive workload and large transaction size. The protocol has been implemented in a key-value storage system called ALOHA-KV, and has been shown experimentally that it can process around 233 million operations per second on 15 servers when transactions contain thousands of operations. Compared to RAMP, ALOHA-KV achieved much higher throughput for large transactions, despite guaranteeing stronger transaction isolation.

Chapter 4

Scalable Serializable Transaction Processing Using Functors

4.1 Introduction

The previous chapter presents *Epoch-based Concurrency Control* (ECC) for high performance serializable distributed read-only and write-only distributed transactions. ECC combines multiversioning and timestamp ordering, and makes transactions visible at epoch boundaries. It seeks to avoid most forms of coordination between transactions by keeping reads and writes completely separated in time. ECC achieves high parallelism in distributed transaction execution for this restricted transaction type (read-only and write-only). The protocol never aborts transactions due to read-write or write-write conflicts but allows transactions to fail due to logic errors or constraint violations. However, this idea has a clear difficulty to overcome: the common case of a single transaction that does both reading and writing.

A read-write transaction allows the transaction execution for updates (writes to the database) to be based on the states of the database (reads from the database). A read-only or write-only transaction can be regarded as a special case of read-write transactions where the write-set or read-set is empty. ALOHA-KV easily supports read-write transactions with read atomicity (RA) isolation, providing superior performance than RAMP for large transaction size. Using conventional methods, ALOHA-KV is able to support serializable read-write transactions by adding a scheduling layer (e.g., lock managers) for concurrency control. However, in that case, the scheduling layer may become a performance bottleneck that hurts parallelism when transactions are under contention.

This chapter proposes a novel paradigm of serializable transaction processing using *functors*, which conceptually resemble *futures* in modern programming languages. A functor is a placeholder for the value of a key, which can be computed *asynchronously* in the future *in parallel* with other functor computations of the same or other transactions. With multiversioning in ECC, the functor computations only rely on accessing historical versions, and so the traditional locking mechanism is not needed for concurrency control. Functors elevate epoch-based concurrency control to a new level: supporting serializable distributed read-write transactions. This combination of techniques keeps the desirable merit: never aborts transactions due to read-write or write-write conflicts, but allows transactions to fail due to logic errors or constraint violations. Using functor-enabled ECC, a read-write transaction is executed in two phases: a *write-only phase* that uses a write-only transaction to store a collection of functors, each of which represents the method for computing a result for a key (e.g., overwriting or incrementing its value); and a *computing phase* that determines the outcomes of the functors asynchronously. ALOHA-DB, a scalable distributed transaction processing system, is implemented using functor-enabled ECC. Experimental results demonstrate that the performance of the system on the TPC-C benchmark with distributed read-write transactions is nearly 2 million transactions per second over 20 eight-core virtual machines, which outperforms Calvin [80, 92, 93], a state-of-the-art transaction processing and replication layer, by one to two orders of magnitude.

The technical contributions of this chapter are as follows:

- This chapter proposes *functors*, which can be used in transaction processing. This chapter presents the structure of functors and the paradigm of transaction processing based on functor computing, which allows a finer level of concurrency control than transaction level.

- It presents an extension of ECC — functor-enabled ECC — for processing distributed read-write transactions. Functor-enabled ECC achieves high parallelism in transaction processing: no transactions are aborted due to conflicts. Using functor-enabled ECC, the transaction execution does not fully rely on deterministic execution, but allows transactions to fail due to logic errors or constraint violations.

- It describes ALOHA-DB, a scalable distributed transaction processing system implementing functor-enabled ECC. This chapter also presents the evaluation that compares the performance of ALOHA-DB relative to Calvin on the TPC-C benchmark and a YCSB-like microbenchmark. ALOHA-DB fully implements the transaction abort logic of TPC-C NewOrder transactions following the benchmark specification, which is not supported in the open-source Calvin implementation [79]. The results show that

ALOHA-DB outperforms Calvin by 1-2 orders of magnitude in terms of throughput in TPC-C experiments, while also maintaining lower latency.

The rest of the chapter is organized as follows. The ALOHA-DB architecture and design are discussed in Section 4.2, which highlights the difference from the ALOHA-KV presented in the previous chapter. Section 4.3 presents the design of functors, focusing on how the functors are used in transaction processing. Then, the chapter presents the functor processing in ALOHA-DB. The evaluation comparing ALOHA-DB and Calvin follows.

4.2 Architecture and Design of ALOHA-DB

ALOHA-DB is a scalable multi-version in-memory transaction processing system that supports serializable distributed transactions across multiple data partitions. Using a combination of ECC and *functors* (detailed in Section 4.3), the system avoids most forms of conflicts among transactions. This section describes the system design of ALOHA-DB and focuses on the novel design points as compared to ALOHA-KV [41], which does not support read-write transactions.

First, this section presents the architectural and design changes required to support read-write transactions in ECC. Second, to accommodate a generic transaction workloads and simplify the epoch advancing design, this chapter proposes an optimization that unifies epoch types to replace the two epoch types (read and write) used in ALOHA-KV. Next, this section describes a method of mitigating the performance penalty due to stragglers. The remainder of this section then presents the storage format for data items and functors.

4.2.1 Architecture

ALOHA-DB is optimized for deployment in a private data center with a high-speed network. Although not necessary for correctness, good network performance and predictability (e.g., low jitter) help the system to achieve high throughput and low latency. Specifically, the network latency helps to reduce the epoch switch time, during which no transaction can be started, and benefits clock synchronization among servers when NTP protocol is used. The architecture of ALOHA-DB, illustrated in Figure 4.1, comprises a collection of servers and an epoch manager (EM). From the transaction processing perspective, the functor computing layer is built on top of the read-only and write-only transaction processing layer, which is derived from the ALOHA-KV. Each server performs the functions of both backends

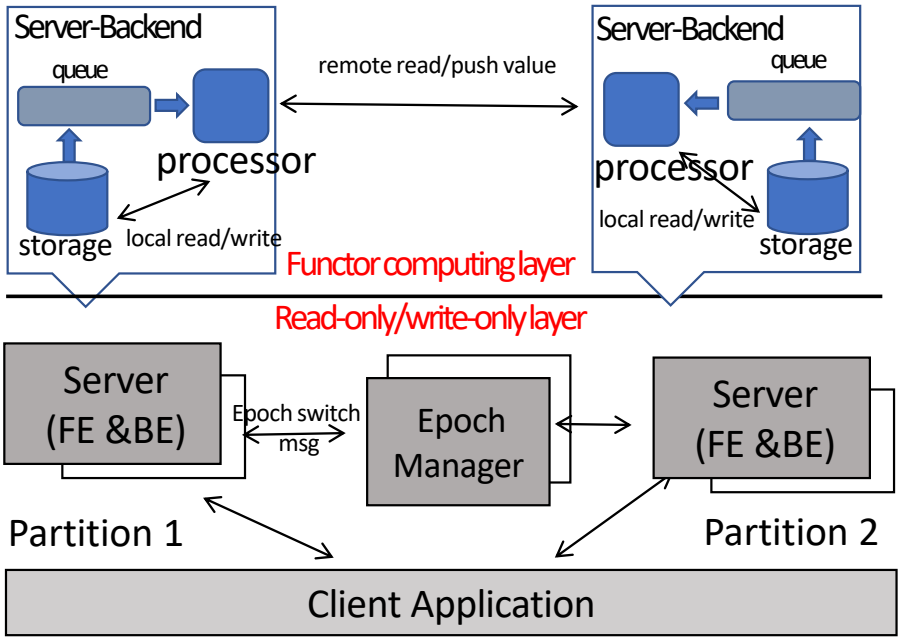


Figure 4.1: Illustration of the system architecture of ALOHA-DB.

(BEs) and frontends (FEs), as in ALOHA-KV. To facilitate understanding the relationship between ALOHA-DB and ALOHA-KV, this chapter uses the terms FE and BE also while describing the system, with the understanding that both terms refer to the same server process.

The EM communicates with all the FEs to control epoch changes by granting and revoking *authorization*, and thus determines when the FEs are able to start executing a given transaction. An FE accepts transaction requests from clients, and acts as a transaction coordinator: it starts transaction execution during the correct epoch, generates a timestamp for each transaction, translates transactions to functors (described later in this chapter), communicates with the partitions, and determines the outcome of the BEs. A BE stores the data items in one partition of the database and serves requests from FEs to read and write these items or functors. BEs also compute functors asynchronously using a component called the *processor*. Whenever there is a functor that is ready to be computed, the BE will push it to a *queue* that will be pulled by the processor. To compute the functors, processors may need to read remotely from other BEs or push data to them. Further details of functor computing are provided in Section 4.3.

ALOHA-DB relies on main memory storage for performance, and therefore depends

crucially on appropriate fault-tolerance mechanisms to protect against data loss and maintain system availability in the event of a server failure. ALOHA-DB is able to leverage the fault tolerance strategies of replication, logging, and checkpointing described in Section 3.5 to achieve reliable epoch switching and to avoid data loss in the presence of a single crash failure.

4.2.2 Unified Epochs

Taking advantage of the fact that reads of historical versions can be processed at any time in ECC, ALOHA-DB unifies the two epoch types described in Chapter 3. In the unified epochs (write epochs), write-only transactions and reads accessing old historical versions can be processed at any time. For a read-only transaction requesting the latest version, this work adopts an optimization that transforms the transaction to an equivalent read-only transaction for a historical version.

In ALOHA-DB there are only a series of write epochs in the system. When an FE receives a read-only transaction for the latest version, the FE assigns a timestamp t to the transaction in the write epoch, and delays processing the transaction until the next write epoch begins. Then, the read-only transaction is processed as a read of historical version t. Informally speaking, the read-only transaction is processed as if it happens at t, but it never conflicts with any write transaction within subsequent write epochs.

By eliminating read epochs, ALOHA-DB allows writes to execute faster because there is no longer any read epoch that might block write transactions, while the latest version read latencies may increase because an FE will always delay such read transactions. However, as described in Section 4.3, a read-write transaction in ALOHA-DB begins reading keys in the read set only after the epoch of the write-only phase completes, and so no additional waiting is required in that case. Moreover, the penalty on read latency for this optimization is bounded by the epoch duration length, which may be tolerable by users when a small epoch duration is used.

Figure 4.2 illustrates an example of ECC with unified epochs. In this example, the epoch switch mechanism is controlled by the EM, as in Section 3.3. The EM grants the FEs authorizations, and the start and end of the validity period are indicated by vertical dashed lines in the figure. It shows that transactions 1–3 proceed as follows:

- Transaction 1, a write-only transaction, can be executed during the validity period.

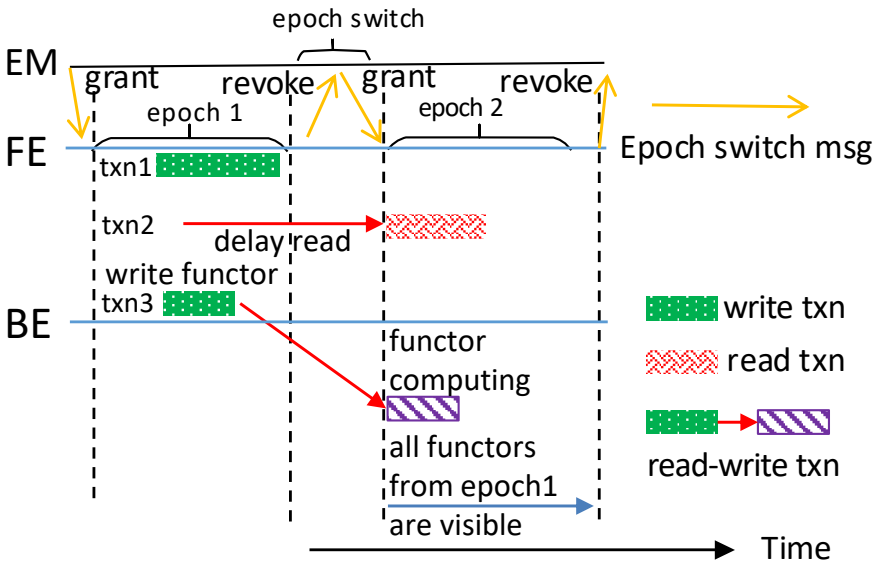


Figure 4.2: Illustration of unified epochs in ALOHA-DB.

- Transaction 2, a read-only transaction accessing the latest version, is assigned a timestamp indicating the current version of the data. In the next epoch, the read transaction will be processed as a historical read for the assigned timestamp.
- Transaction 3, a read-write transaction, has a write-only phase similar to transaction 1 in which it writes the functors to the BEs, and a functor computing phase that may include historical reads similar to transaction 2 using the timestamp assigned in the write-only phase.

4.2.3 Avoiding the Side-Effects of Stragglers

A straggler may delay the epoch advancing and degrade the overall throughput, when it prevents an FE from revoking an authorization for a long time. As discussed in Section 3.4.3, the systems running ECC are unlikely to suffer from stragglers in the absence of software/hardware anomalies. Though ALOHA-DB introduces an extra phase of functor computing in transaction execution, functor computing occurs asynchronously outside of the epochs.

Stragglers remain possible in anomalous cases, and for this reason this subsection presents an optimization that avoids the situation where one straggler prevents all FEs from starting

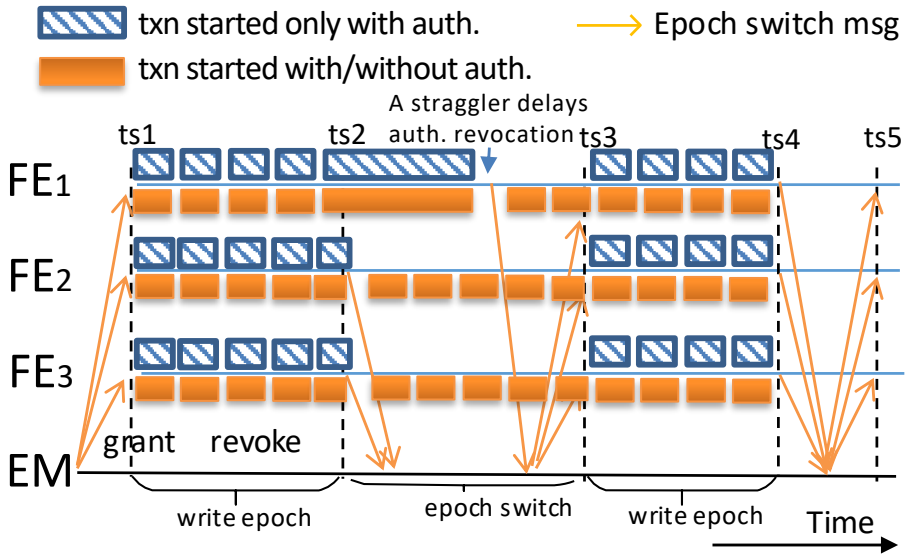


Figure 4.3: Avoiding straggler side-effects by allowing transactions to start without authorization in ALOHA-DB. This figure illustrates execution under two different protocols: transactions started only with authorization (colored blue in the example) and transactions started once the previous epoch completes (colored orange in the example).

the next epoch. It works as follows: FEs can start executing transactions immediately after the authorization is revoked (even without any authorization), as illustrated using the orange bricks (below FE lines) between $ts2$ and $ts3$ in Figure 4.3. These transactions become visible together with transactions started in the following epoch (between $ts3$ and $ts4$). However, this optimization must guarantee that timestamps generated without authorization are smaller than the finish timestamp of the next epoch ($ts4$), as otherwise serializability may be violated. This can be guaranteed by requiring that a transaction without authorization receives a timestamp not exceeding the sum of the previous epoch's finish timestamp ($ts2$) and the duration of the next epoch.

4.2.4 Multi-version Storage

ALOHA-DB stores key-functor pairs in a hash-partitioned distributed table. A concrete value of a key is the final form of a functor (see details in the next section). The functors are versioned to support historical queries, as well as to enable multi-version concurrency control. Figure 4.4 shows the layout of the versions for one key. For each key, the functors

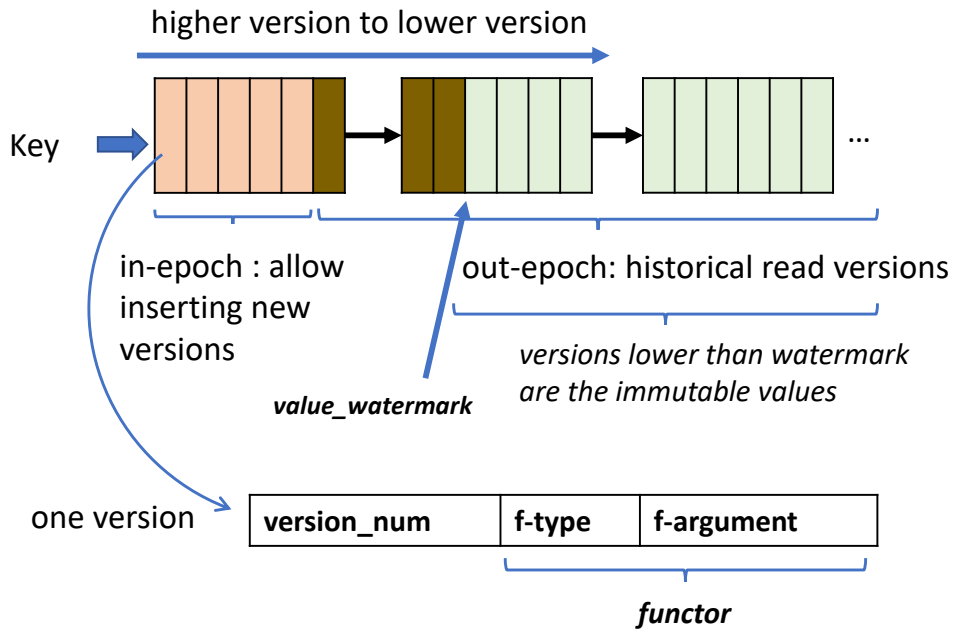


Figure 4.4: Illustration of the storage multi-versioning layout for one key in ALOHA-DB. The linked arrays are used to store the versions of one key.

are organized in a logical list ordered by version, implemented as a linked list of arrays. Each *version record* comprises a version number and a functor. The ordered versions favor accessing the latest version not exceeding a given version number and computing functors for a key in ascending order of versions. As explained in Section 4.2.2, the reads in ALOHA-DB are all historical reads, and the writes are assigned a version equal to their timestamp. As a result, the versions are inserted in nearly sorted order, and so ordered versions are maintained easily.

The API functions *Put* and *Get* are used for accessing the storage layer. A *Put* invoked on a new version of a functor requires the version number to be within the epoch validity period. A *Get* returns the latest version of a key's value not exceeding the requested version. All reads in ALOHA-DB only access historical versions which are less than the epoch start timestamp. Thus, the versions for each key are naturally divided into the *in-epoch category* and the *out-epoch category* by the epoch start timestamp. Versions within the in-epoch category are not visible for reading; versions in the out-epoch category are immutable except that functor computing may replace the functor with its final value. *Get* triggers the

functor computing if the functor to be read is not a final value, and replaces this functor with its final value.

Each key also maintains a special version number called the *value watermark*, below which all the versions are the final value after the functor computing phase (detailed in the next section). Accessing a version below the value watermark needs no synchronization because these versions are immutable. In the implementation, the system uses a lock-free data structure to allow multiple threads to read the storage concurrently.

4.3 Functors

The functor is the core contribution of this chapter that allows ECC to support read-write transactions with low overhead. Functors resemble *futures* [63] in programming language research, which are objects used to represent the future result of asynchronous computations. With ECC, functors enable transactions to first write operator placeholders without any contention in write epochs, and then compute the outcomes of the operators asynchronously and in parallel after the write epoch when the order of transactions is fixed. Functor computing only reads historical versions, thus no locking mechanism is needed on keys when multi-version storage is used. In contrast to other mechanisms that use transaction-level or partition-level concurrency control, functor-enabled ECC uses functors as placeholders for values in the write set, and the functors of a transaction are computed independently and in parallel. Thus, while the basic ECC mechanism provides transaction atomicity and transaction ordering, functor-enabled ECC further allows a key-level concurrency control scheme for read-write transactions that enables high parallelism even under contention.

The design of functors is detailed in the remainder of this section. First, the section presents how a read-write transaction is handled in ALOHA-DB with the help of functors. Then, this section presents how functors are represented and transformed from a transaction. Last, the method to compute the value of functor is discussed.

4.3.1 Transaction Lifecycle

Transaction Model

From the client's point of view, the transaction model is similar to Calvin [93]. This section assumes that transactions are submitted "one-shot" (i.e., non-interactively) from clients and processed by invoking *stored procedures* at servers. The model of stored procedures

is commonly used in previous works [56, 70, 86, 93, 95]. This model does not intend to cover the interactive serializable distributed transactions, because interactions with clients over the network may significantly increase transaction processing time and may slow down overall throughput of the system, especially in contended cases.

A transaction is expressed as a read set and a write set (of keys), as well as a set of arguments supplied by the client. The keys accessed by a transaction must be known ahead of time, which is a restriction also present in Calvin. However, ALOHA-DB does not have this restriction for read-only transactions for historical versions, which are common in analytic workloads. Section 4.4.2 discusses extensions that can work around this restriction. The transactions are executed by the stored procedure in the server side. Conceptually, a transaction stored procedure is a function, in which the function signature (function name, arguments, etc.) denotes the transaction type and the read values based on the read set are provided as inputs.

Lifecycle

The lifecycle of a read-write transaction in functor-enabled ECC includes the following phases:

1. The FE transforms the transaction received from clients (read set, write set, transaction metadata and arguments) to key-functor pairs that are written to the storage system in a write epoch. Each key in the transaction write set will have a functor representing the value of the key after the transaction is executed. All the functors of a transaction share the same transaction version (timestamp) which is assigned by FEs in the ECC protocol described in Chapter 3.

2. In a write epoch, these functors are stored in the BEs as the placeholder for the values of the transaction version. The functors are computed asynchronously after the write epoch, or on-demand at the time of a read. Once computed, each functor is updated with an immutable final value. Each functor can therefore be computed at most once.

3. Based on the client's request, the FEs can acknowledge the transaction execution result once the write-only phase completes, or when the functor computing phase completes. The former acknowledgment option still allows clients to learn the transaction outcome (commit or abort), even if the transaction may be aborted in the functor computing phase. For example, the clients can issue a separate read request to retrieve the result of any of the transaction's functors, because any of the functors will return abort if the transaction is aborted.

4.3.2 Functors for Read-Write Transactions

Interface

A functor is composed of an *f-type* and an *f-argument*. The f-type specifies which *computing handler* to call to compute the functor. The f-argument is a blob whose interpretation is based on the f-type. Table 4.1 shows some examples of f-types and their f-argument representations.

f-type	**f-argument**
VALUE	the literal value of the key
ABORTED	none
DELETED	none
ADD/SUBTR	numerical (e.g., increment value by 1)
MAX/MIN	numerical (e.g., update the value if it is smaller)
user-defined ...	read set and arguments

Table 4.1: Examples of some f-types and their f-argument representations in functors.

The f-type **VALUE** denotes that the f-argument itself is the value, hence no computing is needed for this kind of functor. The f-type **ABORTED** means that this version of the value is aborted, while the f-type **DELETED** is a tombstone of the key, denoting that this key is deleted as of this version. The functors of other f-types require computation that may replace the functor by the value of a key.

Programmers can also create user-defined f-types and the corresponding f-arguments. The user-defined f-type indicates which handler to call for computing the functor. The user-defined f-argument has a functor read set and arguments, which indicate the inputs for the handler. In particular, the functor computing phase requires reading all keys in the functor read set for the latest version not exceeding the functor version. The read set of some functors comprises only the key to which the functor was written, in which case the read set is omitted (e.g., **ADD**, **SUBTR**, **MAX**, **MIN**).

4.3.3 Transforming a transaction to functors

In general, to generate a functor for a key in the write set, one can generate the f-argument by taking the transaction read set and any arguments that influence the result of the key. The corresponding functor handler can be generated in a similar way based on the stored

procedure for processing the transaction. In the current implementation, transactions are transformed to functors manually and automating this process is future work. The following is an example to show the naive method of the transformation. It shows how the transaction stored procedure (TransferMoney), which transfers money from one account to another, can be represented by two functors (TransferFrom and TransferTo). For a given transaction request from clients (assuming overdraft is allowed):

```
TransferMoney: read set {A, B}, write set {A, B}, argument $100
```

may be transformed to the following :

```
<key: A, functor: TransferFrom, read set {A}, 100>
<key: B, functor: TransferTo, read set {B}, 100>
```

or the equivalent functors:

```
<key: A, functor: SUBTR 100>
<key: B, functor: ADD 100>
```

The functor generation phase also may include an optimization to accelerate functor computing: a functor also includes the recipient set, which is the set of keys whose functors' read set includes this key in the transaction. For example, if functors of keys A, B and C all need to read the value of key D, key D may have a functor with recipient set A, B, C. The functor computing of D will send the value of D to the functors of keys A, B, C. This optimization is used to achieve *proactive remote reads* for other functors, meaning that the computing phase of this functor involves *pushing* the latest value of this key to other functors. This design targets speeding up functor computation because it batches several reads for the same version into one read, but it is not required for correctness. Furthermore, the pseudocode and the implementation of the protocol in this chapter do not include this optimization.

After a transaction is transformed to functors, these key-functor pairs are stored in the BEs by a write-only transaction using ECC. This phase is the same as the write-only transaction protocol shown in Chapter 3, and the functor is the "value" to be stored. Thus, for each key in the write set of the original transaction, there is a functor in a BE associated with the transaction version number, as shown in Figure 4.4.

4.3.4 Functor Computing

A transaction is transformed into a collection of functors, and the functors from the same transaction can be computed independently and in parallel, because the functor computing only relies on historical versions. More details of functor computing will be explained shortly.

Computing handler

The functors are computed by handlers in the server backend based on their f-type. In BE storage, a functor is associated with a version of a key (see Section 4.2.4). Functors may be computed by a scheduled thread pool based *processor* in the BE, and may also be computed on-demand at the time when the value is requested by a read, whichever occurs first. Further details regarding functor computing in ALOHA-DB are presented in Section 4.4.1.

Each functor is computed by the *Func* procedure shown in Algorithm 3, which calls the functor computing handler determined by the f-type. Functors that are in their final states with f-type **VALUE**, **ABORTED** or **DELETED**, do not need the computing phase. For other types of functors, the computation begins with deciding the required version for each key in the read set, which is the latest version lower than the version of the functor. Reading is achieved by calling the *Get* function with a version number one less than the version number of the functor, which retrieves data from the multi-version storage of the corresponding partition. This is done only after the write epoch in which a functor was written finishes. Thus, the functor computing phase is able to determine the outcome of any transaction with a lower version number whenever a "reads-from" dependency exists, without contention because the order of historical versions has been fixed and each transaction writes a unique version. Furthermore, functor computing only accesses lower versions (historical versions), which can be read without synchronization if they are final values. If the lower version is a functor that requires computation, the reading thread will compute that functor first, and then update the functor to its final value (line 21).

After reading the values of keys in the read set of the functor, the handler corresponding to the f-type is called. The values read as well as the f-argument are used as inputs to the handler procedure. The output of the handler is used to update the functor, and the functor is updated with the final value at most once. However, in the case of failure recovery, the final value of a functor is recovered by replaying the logs, which will compute the functor again. Thus, the handlers must be deterministic and produce the same outcomes with the same inputs.

Arbitrary Abort

In contrast to deterministic transaction scheduling that must ensure transactions never abort, ECC allows a transaction to abort either in the in-epoch phase or the functor computing phase. In the former case, the FE as the transaction coordinator can send a second round of messages to abort the transaction if any partition fails. This case of abort is enabled by the atomic commitment protocol that is discussed in Chapter 3. In the latter case, the functor computing can decide to abort the transaction as the output, for example, due to an error condition (e.g., insufficient funds for debit). In that case, any keys that influence the abort decision must be in the read sets of all the functors, because they influence the result of all functors of the transaction, and all functors must reflect the same abort/commit decision. This case of aborting can be used to support dependent transactions using OCC as detailed in Section 4.4.2.

Functor Computing Examples

The following presents examples of functor computing for three consecutive transactions in Figure 4.5. The figure demonstrates the states before functor computing on the left side, and the right side shows the states after functor computing.

- Transaction $T1$ atomically updates the value of A and B. The functors for these keys are already the final values, and need no further computing.
- Transaction $T2$ is a read-write transaction that atomically deducts 100 from A and adds 100 to B. The functors for key A and key B are **ADD** and **SUBTR**, in which the read set is the key itself and thus is omitted. To compute the functor for A, the computing process first reads the previous value of A, then updates the functor to a final value using the new value. The computing for the other functor is similar.
- Transaction $T3$ is also a transfer transaction with the constraints that the final value for both keys must be non-negative. Otherwise, the transaction should be aborted. The abort happens if and only if the previous value of A is less than the argument — 100. Thus, the key A must be in the read set of both functors. Similarly to $T2$, for simplicity of presentation the read set of a functor omits the key of itself in the figure.

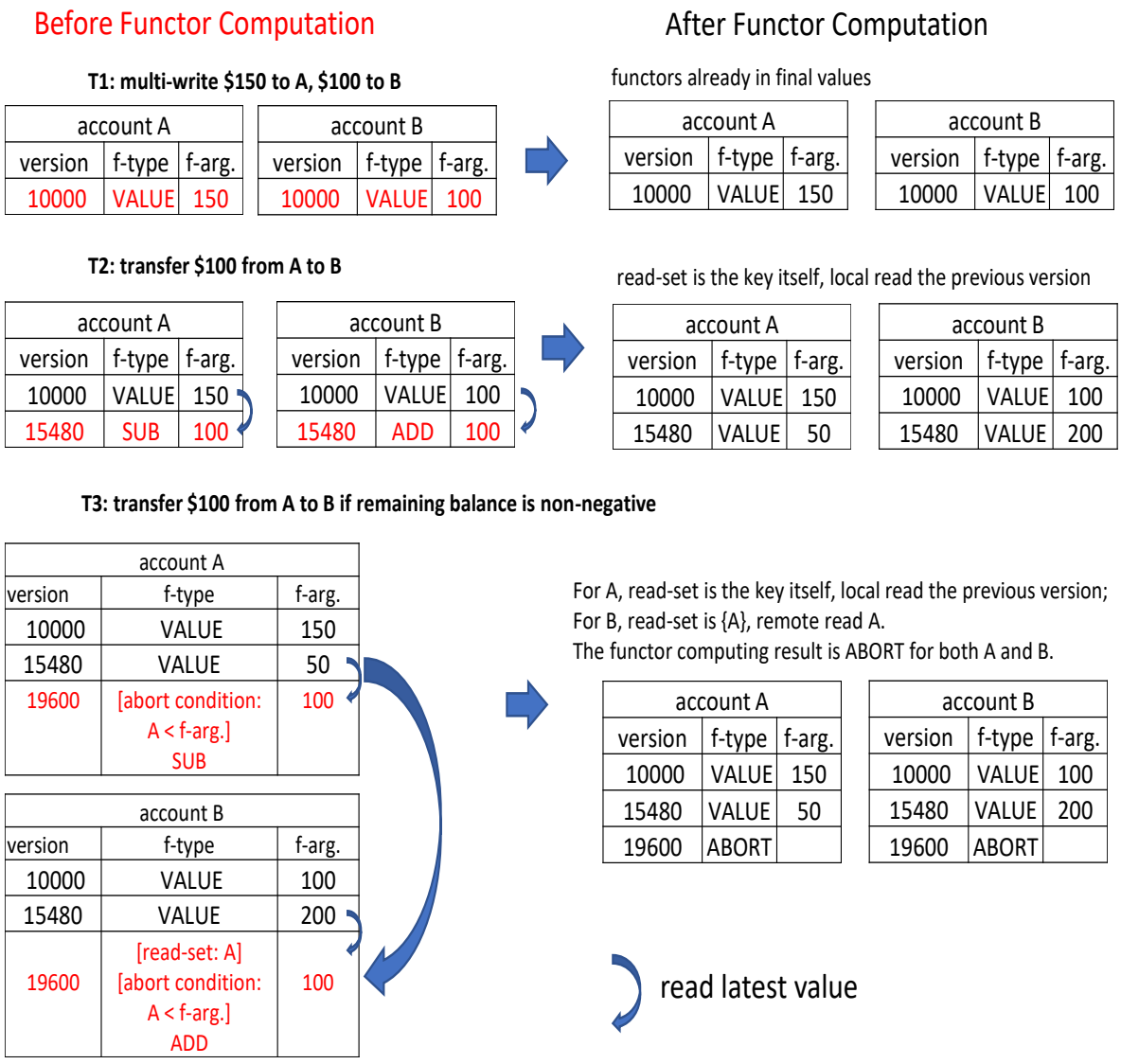


Figure 4.5: Example of three transactions executed using functors over two data items.

4.4 Implementation Details

This section provides more details of the ALOHA-DB implementation. It details the internal functor computing procedure in the system, and presents an optimization that allows ECC to support transactions in which the read sets are not known beforehand.

4.4.1 Functor Processing In ALOHA-DB

Algorithm 3: Functor computing for a specific key k.

– $records[k]$: array of ordered records for key k (initialized with an empty record as version 0), each record is in the form $< v : version, t : f\text{-}type, arg : f\text{-}arg >$
– $watermarks[k]$: value watermark for key k (initial value is 0)

```
1  Procedure Compute(k: key, v: version)
2      w ← watermarks[k]
3      // compute functors from version w to version v for key k
4      foreach record r ∈ records[k] s.t. r.v ∈ [w, v] do
5          if r.t ∉ {VALUE, ABORT, DELETE} then
6              update r using the result of Func (k, r)
7      while w < v do
8          CmpAndSwap(watermarks[k], w, v)
9          w ← watermarks[k]
10 Procedure Func(k: key, r: record)
11     reads : container (map) for values read (initialized as empty)
12     for rk ∈ read set of functor in r do
13         reads[rk] ← Get(rk, r.v - 1)
14     f: handler denoted by r.f
15     return f(reads, r)
16 Procedure Get(k: key, v: version)
17     r : the latest record for k not exceeding version v found by binary search
18     if r.t = DELETE then
19         return ⊥ // denotes deleted key
20     if r.t ∉ {VALUE, ABORT} then
21         Compute (k, r.v)
22     if r.t = ABORT then
23         return Get(k, v − 1)
24     return r.arg
```

Computing the result of a functor requires reading the previous version of any keys in the read set of the f-argument, which means that a higher version functor may depend on one or more lower version functors. If all functors will be computed in the ascending order of versions, the lower versions are always available, but the ordered functor computing scheduling may reduce computing parallelism and hurt performance. ALOHA-DB uses two

design principles to achieve high parallelism in functor processing.

- A greedy functor computing ordering that schedules functor computing in ascending order on a per-key basis. Each key maintains a watermark, namely a version number below which all versions of functors have already been computed. Whenever a functor is computed in ALOHA-DB the watermark will also be updated to the version of the functor by computing all lower versions of uncomputed functors together with this functor.

- Lazily computing the needed lower version functors. When a version dependency relation is found in the functor read set, the functor computing phase will guarantee that the lower versions being depended upon are computed first. Reading a uncomputed functor will trigger the computation of the functor for its final value (see the `Get` procedure in Algorithm 3, which resolves dependencies if applicable).

In the BE, ALOHA-DB uses a thread pool based *processor* to asynchronously compute all uncomputed functors in roughly increasing order of version number. When a new epoch begins, all functors inserted in previous epochs are ready to be processed, and so their meta-data (key and version), which were buffered in the previous epoch, are pushed to a *queue* for the processor to consume. The functors are pushed into the queue by the insertion order of the write-only phase, which is nearly sorted because these write-only transactions are assigned the version number by timestamp.

Algorithm 3 presents the pseudocode for functor computing. For simplicity, in the pseudocode the processor always processes all uncomputed functors of a given key from the watermark to the version obtained from the queue, and then updates the watermark. This design choice is driven by the assumption that many of the functors rely on the previous value of the same key, for example, **ADD** and **SUBTR**. In the implementation, the version obtained from the queue will be processed first if it does not depend on the previous versions of its key, for example, a functor that does not read its own key. This is done to boost processing parallelism, as two of such functors (different versions of the same key) may be computed in parallel when there is no read-from dependency between them.

Processors are also responsible for *remote reading* when the functor needs to read a value from another partition, and for *pushing values* whereby the latest value of a key before the functor version is sent to the functors of any keys in the recipient set (see Section 4.3.3). Pushing a value is a proactive form of reading. As described in Section 4.2, reads may also trigger functor computing on-demand if the value of a functor is not yet available. At read time, computing a functor may involve recursively computing dependent functors with

lower versions. However, this only happens in the case when a read occurs for the functor before the asynchronous processing for that functor has been completed.

4.4.2 Dependent Transactions

Transactions that must perform reads in order to determine the full read set and write set are called *dependent transactions* [93]. This subsection presents two methods for extending functor-enabled ECC to handle dependent transactions. Those two methods of extension do not exclude each other, and dependent transactions can choose one or even both methods (different functors use different methods) based on the transaction characteristic.

Optimistic Approach

ALOHA-DB allows transactions to abort in the functor computing phase. Thus, ALOHA-DB can natively use an *optimistic* approach similar to Hyder [22], which executes transactions by reading from a snapshot and then performs backward validation in the functor computing phase. In particular, a transaction first reads all required keys for some timestamp (e.g., ts_r), determines the write set, and writes all functors with a timestamp (e.g., ts_w). The functors will check whether any values in the read set have changed between the two timestamps, ts_r and ts_w in the example, abort the transaction if so, and commit the transaction otherwise.

In contrast to Hyder, where the validation procedure visits all data versions in the log order whether or not they are relevant to a given transaction, ALOHA-DB functor computing only requires keys in the transaction's read set. This allows multiple transactions to be validated in parallel. As discussed in Chapter 2, OCC has high overhead on retry for high contention workloads, which is the workload targeted by this thesis. Hyder, the same to the ordinary OCC, must abort a transaction that fails validation of any keys in the read set, while in ALOHA-DB only the keys in read set that determinate the write set may cause the backward validation to fail. A detailed performance comparison between Hyder and ALOHA-DB is left as future work.

Key Dependency

For a dependent transaction, the transformation from a transaction to functors that generates a functor for each key in the write set cannot be achieved until the functor computing phase when the functors can read previous versions of keys. To solve the problem without resorting to optimistic concurrency control (OCC), the design defers the

write-only phase for the keys that can only be determined as part of the write set during the functor computing phase. These keys are called *dependent keys*, because they are decided in the functor computing phase of functors belonging to other keys in the same transaction. This chapter refers to these functors as *determinate functors*, and the keys that determinate functors belong to are called *determinate keys*.

Algorithm 4: Functor computing for dependent transactions.

```
  – determinate[k]: the set of determinate keys of key k
1 Procedure Compute_kd(k: key, v: version)
2     if watermarks[k] > v then
          return
3     for dk ∈ determinate[k] do
4         Compute_kd(dk, v)
5     Compute(k, v)
6 Procedure Get_kd(k: key, v: version)
7     Compute_kd(k, v)
8     return Get (k, v)
```

For example, consider a transaction that will write key B only if the value of key A satisfies some condition. This transaction will choose A as a determinate key, record a determinate functor for A in the write-only phase that will write B (dependent key) in the functor computing phase if the value of A satisfies the condition, and store no functor for key B in write-only phase. If B is written, the version number applied to the dependent key B will be the same as that of the determinate functor, because all the writes belong to the same transaction. Thus, whenever calling *Get* on the dependent key B for a timestamp ts, the value watermark of key A must be at least ts to guarantee that all "deferred writes" on B have completed. In other words, for any given version, key A's functors must be computed first before reading the same version for key B, otherwise serializability may be violated. Algorithm 4 shows how the design guarantees that the functor computing phase always computes determinate functors before reading dependent keys for a given timestamp. In particular, the design provides alternative implementations of the `Compute` and `Get` functions. Each read (`Get_kd`) will check the watermark of the determinate keys (line 7), and if some determinate key's watermark is lower than the given timestamp, the determinate key will be computed first.

For performance reasons, the design imposes the restriction that the dependent keys and determinate keys are in the same partition, which is satisfied by the workloads used in this chapter. The design assumes the dependency relationship between keys (denoted

determinate[k] in Algorithm 4) can be obtained by static analysis of all the stored procedures. This is also a restriction of the method of key dependency. Taking as a special case, one can use a special key, namely ANY, as the determinate key for all other keys in the partition. Each dependent transaction is transformed to a determinate functor for the key ANY, and the functor computing phase always computes functors of ANY first. In that case, all the dependent transactions are processed serially in timestamp order before any other transaction. This solution may be appealing when the dependent transactions are infrequent in the workload.

The transactions used in the evaluation of chapter do not use the optimistic approach, but use a determinate functor as described in the experiments setting in Section 4.5.1.

4.4.3 Serializability

This subsection extends the discussion of serializability of ECC in Section 3.6.3 to sketch the proof that functor-enabled ECC also provides serializability.

Lemma 2. *Every history of transactions processed by functor-enabled ECC is one-copy serializable (1SR).*

Proof sketch. Given a history H of committed transactions, which have completed the functor computing phase of the functor-enabled ECC, assign a timestamp to each transaction in the same way as Section 3.6.3 (a read-write transaction is assigned a timestamp as a write-only transaction in the write-only phase). Now arrange the transactions into a serial history S in increasing order of their timestamp, breaking ties arbitrarily for read-only transactions. It follows easily that H and S have the same committed transactions and operations.

This paragraph presents the proof that H and S have the same "reads-from" relationships. For simplicity, this paragraph only consider read-write transactions, and read-only or write-only transactions can use similar proof as they can be treated as special read-write transactions. The proof is by contradiction. Assume some transaction T does not read the highest version ts' of a key that does not exceed its own timestamp ts, where $ts' < ts$. Then this transaction T is executed before the transaction T' that creates the version ts', otherwise T should read T'. Let E' denote the epoch in which the write-only phase of transaction T' is executed. Let E_w denote the epoch in which the write-only phase of transaction T is executed. Let E_c denote the epoch in which the functor computing phase of transaction T begins to be executed. An epoch *precedes* another epoch when its authorization period finish timestamp is before the authorization period start timestamps

of the other epoch. According to functor-enabled ECC, E_w precedes E_c. If E' precedes E_c then T would have seen T', contrary to the earlier assumption, and so E_c precedes or is equal to E'. Thus, E_w precedes E'. The authorization periods for E_w and E' are disjoint no matter how badly skewed the local clocks of servers are according to ECC. Thus, E_w preceding E' implies that T has a lower timestamp than T' because ts and ts' must be within the authorization period of E_w and E'. This implies that $ts < ts'$, which contradicts the earlier observation that $ts' < ts$.

As a result, H and S have the same "reads-from" relationships, and moreover S is one-copy serial because S is a serial history. This implies that H is 1SR. □

4.5 Evaluation

To evaluate the performance envelope of functor-enabled ECC, ALOHA-DB is implemented on top of the ALOHA-KV [41] codebase. It is programmed in C++ using the popular open-source RPC framework fbthrift [36]. The experiments run both the TPC-C, the scaled TPC-C (detailed in Section 4.5.1), and the YCSB-like microbenchmark on ALOHA-DB and Calvin [80, 92, 93], a state-of-the-art high-performance transaction processing system for distributed transactions. The experimental results show that ALOHA-DB outperforms Calvin by one to two orders of magnitude in terms of throughput, and also achieves lower latency. ALOHA-DB fully implements the aborting requirements for TPC-C NewOrder transactions as required in the benchmark, in contrast to the open-source implementation of Calvin. ALOHA-DB outperforms Calvin in terms of throughput under various database partitioning strategies, indicating that the system has lower overhead for distributed transactions even under high contention scenarios. Specifically, the results show that ALOHA-DB achieves nearly 2 million distributed NewOrder transactions per second across 20 servers, which is 13–112× faster than Calvin.

4.5.1 Experimental Setup

Workload

TPC-C [94] is a standard benchmark for online transaction processing (OLTP), which simulates the activity of a wholesale supplier. The benchmark has 9 tables and 5 types of transactions. Similarly to prior work [12, 52, 93], the experiments based on TPC-C perform only NewOrder transactions and Payment transactions because the focus is on distributed

read-write transactions and TPC-C performance is dominated by these two transactions, which comprise 88% of the total workload. The NewOrder transaction first reads its warehouse, district, and customer records, then it updates the district record, new-order, and order tables. Next, the transaction updates 5 to 15 items in the stock table. 1% of NewOrder transactions must be rolled back as a result of an unknown item number. The Payment transaction first updates the payment amounts for the associated warehouse and district records, then it updates the customer table and inserts a new record into the history table. The multi-partition Payment transaction should access two partitions, because the customer will belong to a remote warehouse that is on another server. ALOHA-DB also supports OLAP workloads in parallel with updates thanks to multi-version storage, but the evaluation of mixed OLAP and OLTP workloads is outside the scope of this thesis.

Scaled TPC-C [70], modifies the partition-by-warehouse approach, where each server holds all data related to one or more warehouses, by partitioning data within one warehouse. This workload is more suitable for stress-testing the performance of distributed transactions. The Scaled TPC-C treats the database as a single warehouse, partitions the database by item and district, and simulates the behavior of a large warehouse that spans many servers. For example, [70] reports that Amazon uses around 100 warehouses to serve more than 300 million global customers. That means a warehouse must handle a large number of customers, which should be distributed on multiple servers. The experiments implement and evaluate the scaled benchmark, denoted by *scaled TPC-C*, as well as the conventional partition-by-warehouse benchmark that is used in Calvin papers [80, 93], denoted by *TPC-C*. Payment transactions are only implemented in TPC-C, because the Scaled TPC-C partition strategy in [70] removes the w_ytd field from the warehouse table, which is needed for the Payment transaction. For a fair comparison, (non-scaled) TPC-C transactions are generated in the same way as in Calvin: a distributed transaction always accesses a second warehouse that is not on the same server as the first.

YCSB [28], is a benchmark designed for Internet-scale storage systems. YCSB does not include a standard read-write transaction workload. The YCSB-like microbenchmark implemented in Calvin is chosen for the evaluation, because it has a tuning knob (contention index) that can accurately specify the contention on a partition for a distributed transaction, and it is the main workload used in previous Calvin evaluations [80, 93]. We reproduce the microbenchmark implementation of Calvin [80] for experiments with ALOHA-DB. In the microbenchmark, each server contains a database partition consisting of 1M keys. On each partition, the data items are divided into "hot keys" and "cold keys". Each transaction reads 10 keys then updates the keys by increasing the value by 1, and accesses exactly one hot key at each participant partition. A distributed transaction touches two partitions. When each partition has K hot keys, the contention index (CI) is defined as $1/K$. Thus,

two concurrent transactions executed on the same partition have a probability of $1/K$ to collide. For example, a CI value of 0.01 denotes that each transaction executed on a partition must access 1 of the 100 hot keys.

Unless specified otherwise, all transactions in the experiments are distributed transactions, which update data items on more than one server.

Comparison of Systems

Unless specified otherwise, ALOHA-DB uses a 25ms unified epoch duration, which provides a performance balance between write throughput and read latency in the experiment environment presented in this section. As described in Section 4.2.2, unified epochs cause all reads to be processed as reading a historical version, and to be executed in parallel with write transactions. In comparison, Calvin's sequencer batches requests in epochs of 20ms, because it is found that Calvin has no significant throughput improvement with 25ms epochs but only longer latencies, and shorter epochs result in lower throughput under the same configuration. Both systems are configured for in-memory storage, and fault tolerance is disabled by default to follow the same convention as in the Calvin papers [80, 93].

ALOHA-DB fully implements the requirement that 1% of NewOrder transactions must abort [94]. Specifically, the aborted transaction includes an item that cannot be found in the corresponding partition, while other partitions process the transaction as usual in the first phase of the transaction commitment protocol. The transaction coordinator (FE) must issue the second round of messages to abort the transaction and roll back the processing on the other partitions. In contrast, Calvin's implementation does not support aborted transactions because of its deterministic design [80]. As a result, Calvin is able to pre-assign the order_id efficiently for a NewOrder transaction, whereas ALOHA-DB must assign the order_id dynamically in the determinate functor processing phase (see Section 4.4.2). Specifically, the next_order_id is the determinate key of the Order, NewOrder and OrderLine tables.

For an apples-to-apples comparison, ALOHA-DB submits a batch of transaction requests in each RPC call, similarly to Calvin. This ensures that neither system is bottlenecked on the RPC layer. However, because ALOHA-DB workloads include 1% aborting transactions which need the second round of messages to abort, the epochs complete only after all such transactions finish the second round. This gives Calvin a potential latency advantage since its open-source implementation does not support aborting transactions at all.

Environment

By default, the experiments use eight Amazon EC2 m4.4xlarge virtual machine instances with hyper-threading disabled (8 cores total). The experiments choose such instances because existing Calvin code is optimized for 8-core machines. However, it is believed that ALOHA-DB has the potential for higher performance using more powerful machines, as extra resources are provided. A BE/FE pair is co-located in one process at each host and the EM process shares one of these hosts. The results presented are the aggregate throughput and average latency. The transaction latency measurement in both systems is made in the same way: from when the transaction is issued by the client until its functors (ALOHA-DB) or replicated transactions (Calvin) have completed execution. Both systems focus on server-side latency, and the latency results do not include the final response to the client. There are three runs for each combination of parameters, and variation across runs is indicated using vertical error bars representing the min and max measurement. In most cases, the error bars are imperceptibly small.

4.5.2 TPC-C Experiments

Results: Throughput vs. Latency

This subsection first presents the results for throughput vs. latency of NewOrder transactions in ALOHA-DB and Calvin under both TPC-C and scaled TPC-C in Figure 4.6. For TPC-C experiments, two partition settings are used: 1 and 10 warehouses per host, denoted as **1W** and **10W** respectively. Similarly, **1D** and **10D** denote 1 or 10 districts per host in scaled TPC-C experiments.

In terms of peak throughput, ALOHA-DB performs around 13× (in TPC-C) and 61× (in scaled TPC-C) greater than Calvin. The Calvin-10W peak throughput (close to 60k tps) is comparable to the results in [80], though higher because the experiments presented in this thesis use more powerful virtual machines. However, with fewer warehouses per host or under a scaled TPC-C workload, Calvin suffers a significant throughput drop, while ALOHA-DB's performance under different settings is much more steady. Further comments on this observation follow in Section 4.5.4.

The latency results show that Calvin has higher latency than ALOHA-DB in light workloads, while sustaining much lower throughput. As explained in [80], preprocessing of requests in the scheduling layer contributes to latency in Calvin. The latency in ALOHA-DB also includes the time functors spend waiting for the epoch to finish before starting computing, and so the average latency must be larger than half of the epoch duration.

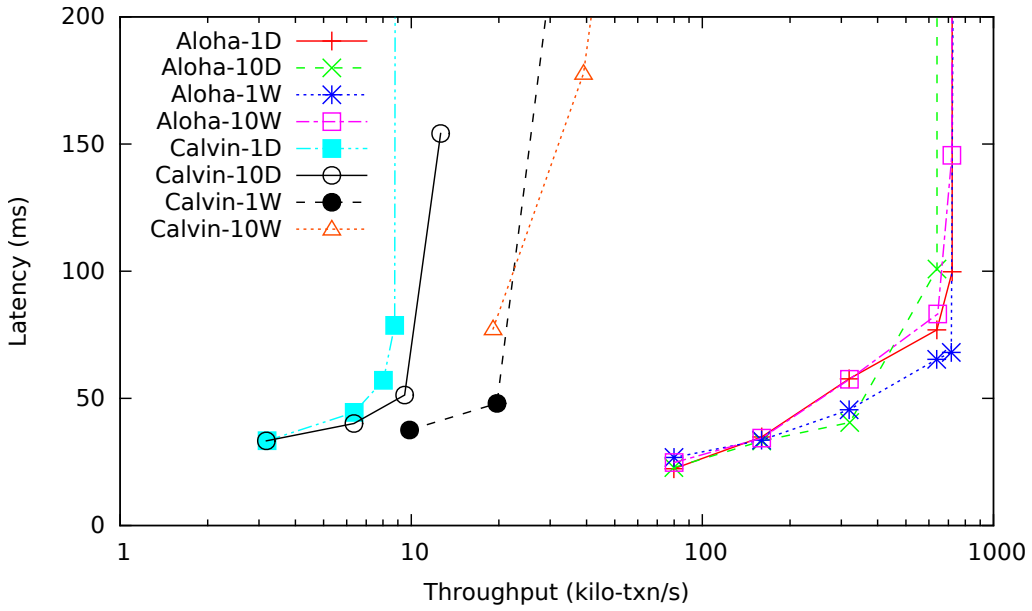


Figure 4.6: Throughput vs. latency: ALOHA-DB and Calvin experiments for NewOrder transactions on eight m4.4xlarge instances. Logarithmic scale used for horizontal axis. **1W** or **10W** denotes 1 or 10 warehouses per host in TPC-C experiments; **1D** or **10D** denotes 1 or 10 districts per host in scaled TPC-C experiments.

Database Partitioning

The database partition strategies affect transaction contention and distributed transaction access patterns. Given the same workload in TPC-C experiments, having fewer warehouses at one host creates more contention for each warehouse. In particular, the Payment transaction has higher contention than the NewOrder transaction, as it updates the warehouse table while the NewOrder transaction updates one of the ten districts within the warehouse. In terms of distribution, a NewOrder transaction in the TPC-C experiments only contacts two partitions, but likely contacts more than two partitions in scaled TPC-C where partitioning is done by item.

Figure 4.7 shows the results for TPC-C NewOrder and Payment transactions with 1 to 10 warehouses per host, denoted by TPCC, and Scaled TPC-C NewOrder transactions with 1 to 10 districts per host, denoted by STPCC. Calvin has 2–4× lower throughput for scaled TPC-C than for TPC-C because more partitions are involved in a transaction. At the same time, there is no significant drop in ALOHA-DB scaled TPC-C throughput performance.

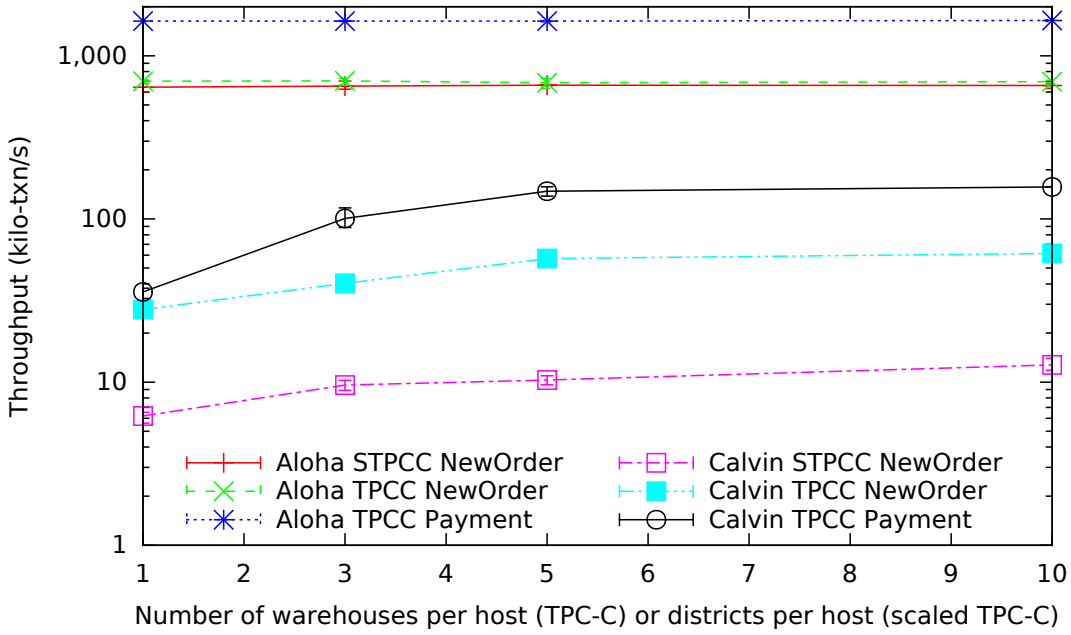


Figure 4.7: ALOHA-DB and Calvin throughput for NewOrder and Payment transactions under various numbers of warehouses or districts per host. Logarithmic scale used for vertical axis.

In the write-only phase of ALOHA-DB, the write-only transactions are executed in batches, and so the distribution of individual transactions across multiple partitions has little impact on the aggregate throughput. Section 4.5.4 will further discuss the explanation regarding the functor computing phase.

Considering various numbers of warehouses per host in TPC-C, Calvin exhibits a performance drop when the number of warehouses decreases, and the drop is more severe when the number of warehouses is small. For example, the throughput of NewOrder transactions in Calvin drops from 40k to 28k when the number of warehouses changes from 3 to 1. The Payment transactions in Calvin begin to suffer a throughput penalty when the number of warehouses is less than 5, because the contention increases on the warehouse table when each host has fewer warehouses. Even though Calvin resolves conflicting transactions by deterministic ordering, it is bottlenecked at accessing the contended data items under high contention, which requires synchronization by the lock manager. Thanks to multi-versioning and timestamp ordering, ALOHA-DB does not use locking for concurrency control.

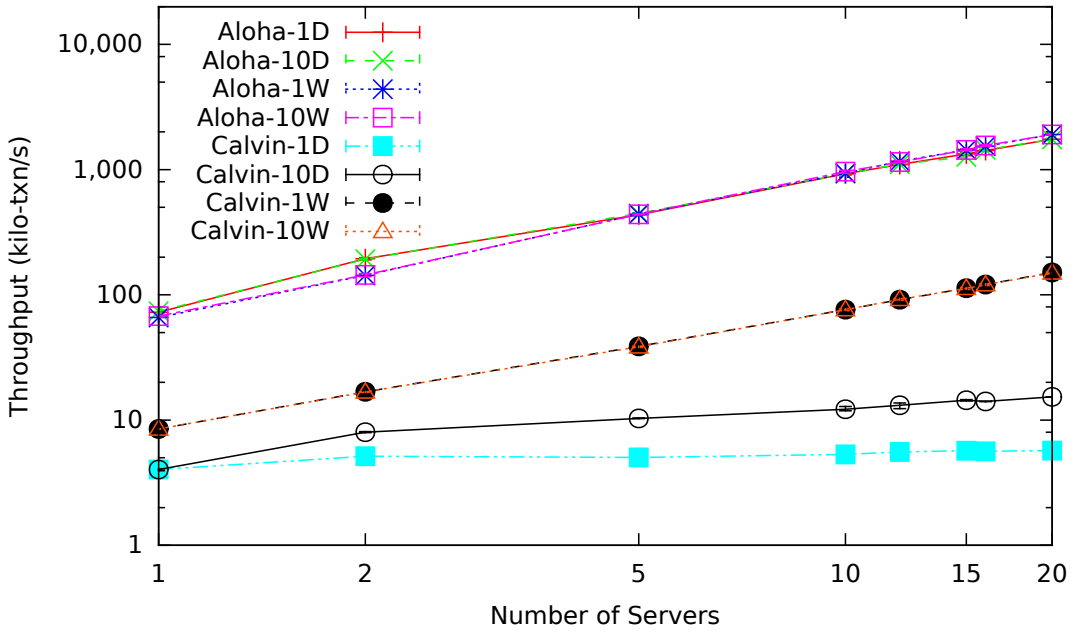


Figure 4.8: ALOHA-DB and Calvin scale-out performance for NewOrder transactions. Logarithmic scales used for both axes.

In comparison, the performance drop under high contention or high transaction distribution is less than 5% in ALOHA-DB, even in the case of 1 warehouse per host or 1 district per host experiments. ALOHA-DB supports a high-performance write-only phase thanks to ECC (see the ALOHA-KV paper [41]), and the functor computing phase uses fine grained (key-level) concurrency control to achieve high parallelism. In the high contention cases, functor computing might also benefit from sequential memory access, when many functors of the same key are processed together (see line 4 in Algorithm 3), while Calvin computes keys belonging to the same transaction together.

Scale-Out

Figure 4.8 presents the NewOrder transaction throughput results using up to 20 servers. Nearly linear scalability is observed with the exception of Calvin with Scaled TPC-C. ALOHA-DB achieves up to around 2 million transactions per second in total, which is 13–112× faster than Calvin. For Scaled TPC-C, the Calvin does not scale well because a transaction likely needs to contact more partitions as the number of servers increases,

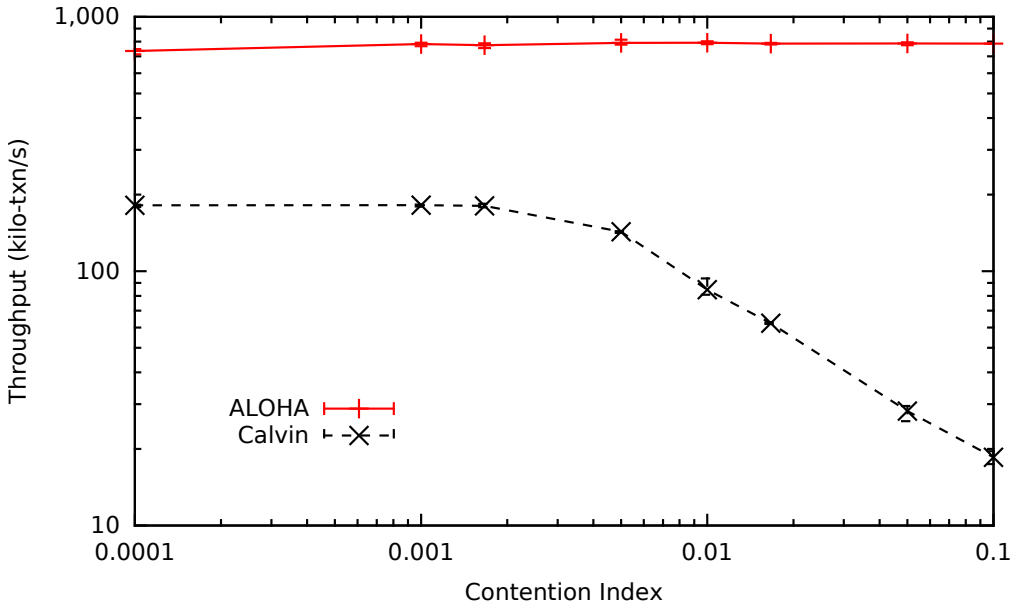


Figure 4.9: ALOHA-DB and Calvin microbenchmark performance under various values of the contention index. Logarithmic scales used for both axes.

while it only contacts two partitions in TPC-C. For the same reason, Calvin TPC-C result is nearly linear. In contrast, ALOHA-DB's functor processing overheads do not increase significantly when a transaction needs to contact additional partitions, as explained in next subsection.

4.5.3 Microbenchmark Experiments

Skewed Workload

Typically, distributed transaction processing using conventional transaction-level concurrency control suffers under a skewed workload, because transactions are forced to wait for the contended keys and coordination involving remote servers (e.g., 2PC) is usually slow. Previous works [52, 80] have already shown that Calvin, using partition-level concurrency control, outperforms conventional transaction-level concurrency control for high contention cases. However, Calvin still suffers under skewness within a partition when the lock manager is contending for hot keys [80].

Figure 4.9 demonstrates the throughput of ALOHA-DB and Calvin under various contention index settings. When the CI is less than 0.002 (600 hot keys per partition), Calvin still performs around the peak throughput. However, the throughput begins to drop with a more skewed workload. In contrast, ALOHA-DB does not suffer a significant throughput drop in these settings. In the highly skewed case (CI is 0.1, 10 hot keys per partition), each partition processes nearly 97k txn/s on average in ALOHA-DB.

Further experiments are conducted to break down the latency of low and high contention cases (CI 0.0001 and 0.1), under a light workload (5% of peak throughput). Figure 4.10 shows the percentage of time used in different stages of a transaction lifecycle. In ALOHA-DB, the *Functor installing* stage measures the duration from when a transaction is issued to when a functor is installed in the BE (the epoch may be unfinished); *Waiting for processing* describes the duration from when the functor is installed to when the functor is retrieved by processors; *Processing* time denotes the stored procedure running time for the functor computing. For Calvin, *sequencing* stage denotes the duration from when a transaction is issued to when the partition scheduler begins to process the transaction (comparable to the functor installing and waiting for processing in ALOHA-DB); *locking and read* duration includes the time of locking all required locks and reading keys in the read set; *Processing* denotes the stored procedure running time. Note that the Waiting for processing stage and sequencing stage both needs to wait for the completion of the epoch, thus depend on the epoch duration. In both systems, the processing stage takes the minimum time, and the largest part is spent completing the epoch. However, the latency of Calvin is more sensitive to high contention, as in the high contention case each transaction spends more time on average in the locking phase.

Varying Transaction Size

To study the benefit of the key-level concurrency control under high and low contention, the performance of single-partition transactions is measured, where Calvin does not have the side-effect of redundant execution. For each server, the high contention setting uses 1K keyspace and CI 0.01; the low contention setting uses 1M keyspace and CI 0.0001.

Figure 4.11 shows the throughput of ALOHA-DB and Calvin under various transaction sizes. The figure presents the total throughput in units of operation per second, which is calculated as transaction throughput times transaction size. The results demonstrate that ALOHA-DB irrespective contention, and Calvin under low contention, have stable throughput numbers while transaction size changes. In these cases, the throughput of operations has saturated the storage systems, thus the transaction throughput is the reciprocal of transaction size as the operation throughput is nearly flat, which is the

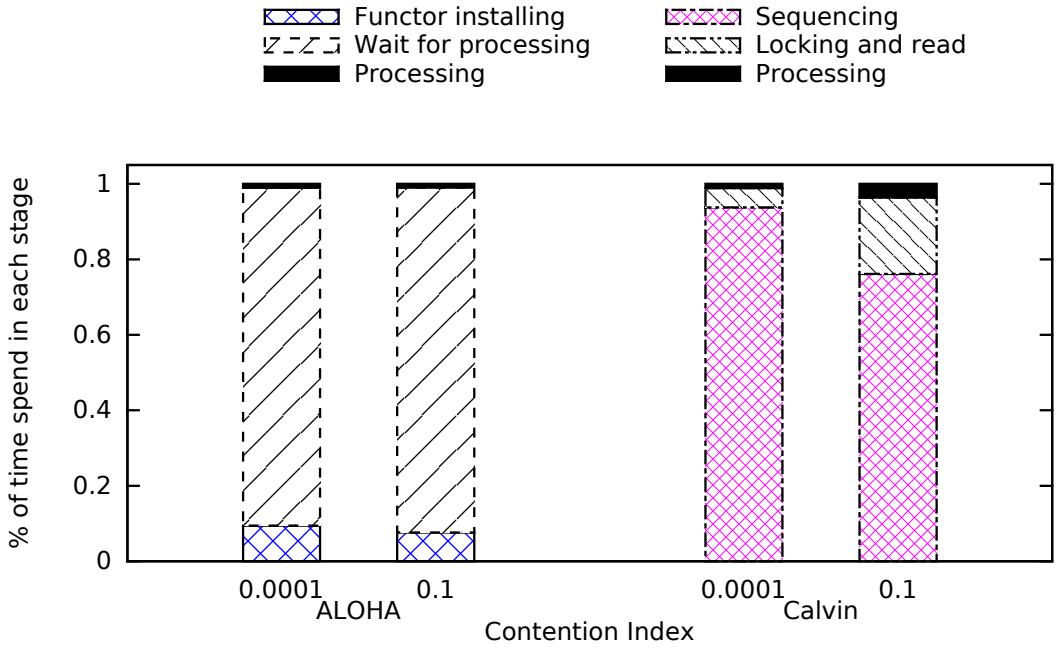


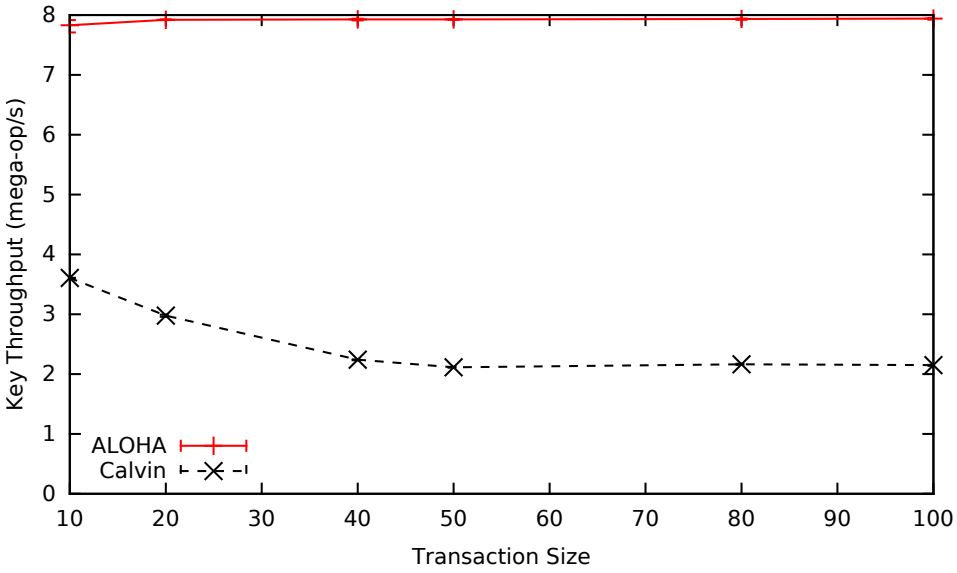
Figure 4.10: Latency breakdown: latency of different stages of a transaction lifecycle in ALOHA-DB and Calvin under low and high contentions.

product of transaction throughput and transaction size. It is noticeable that Calvin, with a single-threaded locking mechanism, has slower throughput even in the low contention case than ALOHA-DB, which uses decentralized timestamp generation for timestamp ordering.

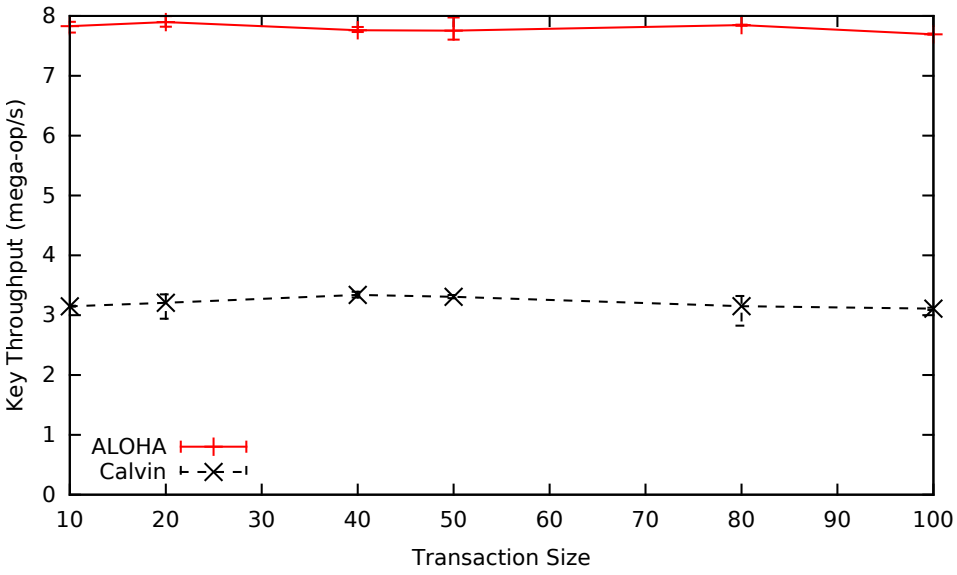
Under high contention, Calvin's throughput drops when large transaction size is used, while ALOHA-DB does not. This occurs because Calvin holds locks for all keys of a transaction during the transaction execution (larger transaction size increases the chance of being blocked by some hot keys). ALOHA-DB uses a key-level concurrency control approach that processes each key individually, thus permits more parallelism even in the case of large transaction size.

Varying Epoch Durations

Figure 4.12 presents the latency of ALOHA-DB and Calvin for various epoch durations, under medium contention (CI 0.001) and a light workload. In the results, the average latency is nearly linear with respect to the epoch duration for both systems, although the slopes are different. For ALOHA-DB, functors need to wait half of the epoch duration



(a) High contention experiments: using 1K keyspace and CI 0.01.



(b) Low contention experiments: using 1M keyspace and CI 0.0001.

Figure 4.11: Throughput of single-partition transactions under various transaction size.

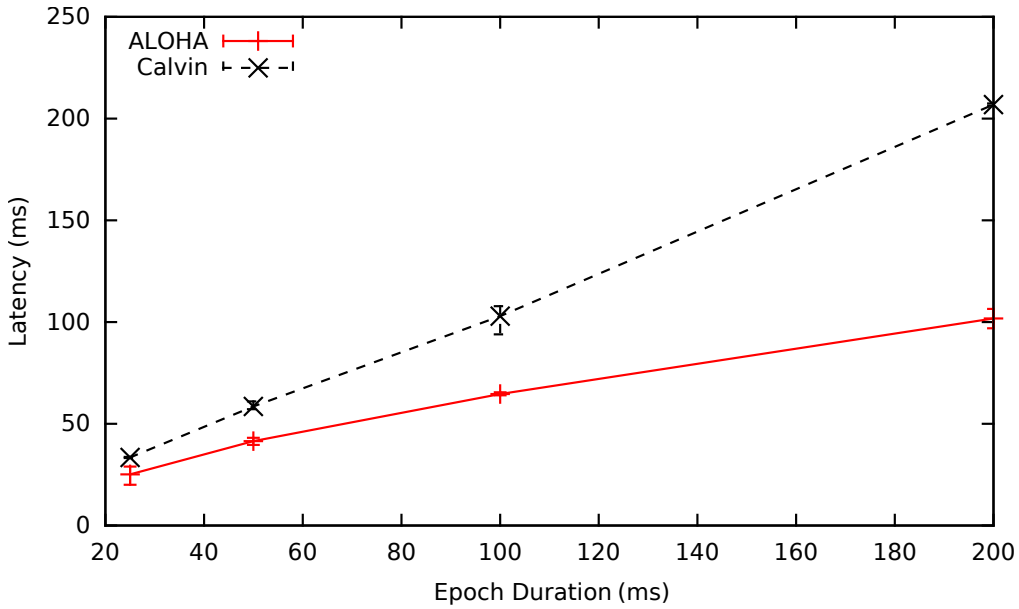


Figure 4.12: ALOHA-DB and Calvin latency under various epoch durations.

on average after they are installed in the BE, thus the linear slope is close to 0.5. The open-source Calvin implementation generates most of the transactions at the beginning of the epoch, when the number of generated transactions quickly reaches the target threshold of transaction number per epoch. This is a side-effect of the workload implementation in Calvin when the target throughput is low, while workload generator in ALOHA-DB considers both target throughput and epoch duration. Thus, in the figure we observe that the linear slope is close to 1 for Calvin.

4.5.4 Discussion

This subsection summarizes some of the design characteristics of the two transaction processing systems under consideration in the experiments, and their performance consequences, to elucidate the experimental results.

- Calvin uses partition-level concurrency control. Calvin first replicates a transaction to all involved partitions. Each of these partitions reads all the values in the read-set, *redundantly* executes the same stored procedure on each partition, but only writes the keys belonging to this partition. However, as all partitions read the same read

set, some remote reads result in wasted work for a partition where the key read has no influence on the writes performed in this partition.

- The write-only phase in ALOHA-DB has concurrency control overhead close to eventual consistency thanks to the low overhead ECC. This point was established in [41] by showing that ALOHA-KV has performance close to the baseline of no concurrency control. This is because write-only transactions in write epochs require almost no concurrency control and the epoch switch only takes a small portion of the execution time.

- The ALOHA-DB functor computing phase performs key-level concurrency control. Multi-versioning allows functor computing to avoid the locking mechanism, but a functor cannot be computed if a version to be read is not a final value. However, in this case the thread will begin to compute that functor on which it depends, rather than blocking until another thread computes this functor (see line 21 in Algorithm 3). This resembles a rescheduling of the functor execution, and ensures that threads are well-utilized.

In light of the decisions discussed above, the performance differences between ALOHA-DB and Calvin can be explained as follows.

- Multi-versioning and key-level concurrency control in ALOHA-DB provide more parallelism than the single versioned partition-level locking concurrency control in Calvin. Thus, even in the extreme case of 1D or 1W where Calvin may execute transactions one by one in a partition, ALOHA-DB still allows different threads to compute different functors in parallel.

- Each functor is computed only once in ALOHA-DB which avoids the redundant execution that happens in every participating partition in Calvin. This difference becomes more pronounced in the scaled TPC-C experiments, where a transaction may touch 15 partitions when the order includes 15 items on different partitions, in which case the processing is repeated 15 times.

- Functor computing only reads the value it needs, thus avoids all unnecessary remote reads in ALOHA-DB. In both TPC-C and Scaled TPC-C experiments presented in this section, most of the functors only need to read the latest value of the same key, or additional keys in the same partition. For example, in the scaled TPC-C experiment, stock data is partitioned by item id. Using functors, when a partition only needs to update some counters, such as quantity, order_count, year_to_date, and so no remote

read is needed for this partition. However, in Calvin, the same partition must read all keys in the read set, and most of these reads are remote. Thus, ALOHA-DB performs similarly for both TPC-C and scaled TPC-C.

4.6 Summary

This chapter addressed the problem of supporting high-throughput multi-partition transactions. In particular, it proposed a mechanism to support serializable distributed read-write transactions by extending the epoch-based concurrency control (ECC) approach, introduced in Chapter 3, that works on read-only and write-only transactions. The method described in this chapter uses write epochs to record functors, which are objects that represent how to evaluate the corresponding versions of values. A functor is processed either during asynchronous batch processing or at read time, once all versions on which the functor depends are settled. This chapter also presented the implementation of the protocol in a transaction processing system called ALOHA-DB and has evaluated it using the TPC-C benchmark and YCSB-like microbenchmark for read-write transactions. For the TPC-C benchmark, ALOHA-DB was shown to achieve nearly 2 million transactions per second on 20 servers, which is 1 to 2 orders of magnitude faster than Calvin, while supporting lower latency and also allowing transactions to abort due to logic errors.

Chapter 5

Understanding the Causes of Consistency Anomalies in Apache Cassandra

Chapter 3 and Chapter 4 both focus on improving scalability for systems that support serializable transactions. This chapter investigates systems that are on the other side of the consistency spectrum: NoSQL systems that can easily achieve linear scalability with eventual consistency. Using a popular open source eventually consistent storage system, Apache Cassandra [2], as the model, this study explores the possibility of achieving both scalability and consistency by understanding and improving the consistency of NoSQL systems.

5.1 Introduction

This study is motived by the observation of consistency spikes in the staleness time series experiment of Rahman et al. [77]. As shown in Figure 1.1 in Chapter 1, the graph exhibits occasional abrupt spikes of the staleness of values returned by read operations applied to Cassandra—an open-source distributed storage system that supports eventual consistency. Research on the cause of these spikes can help us to understand the service provided by those eventually consistent systems, and further help the service providers to improve customer satisfaction, e.g., targeting a consistency level that may be contracted by service level agreements (SLAs) [91].

The investigation begins with reproducing the spikes observed in prior experiments and then explores the low-level mechanisms that give rise to stale reads. A hypothesis is formulated that the observed spikes are caused by the Java virtual machine's garbage collection (GC) "stop-the-world" (STW) phase, which pauses all application threads. The hypothesis is experimentally tested by correlating the occurrence and duration of the GC pause time against the occurrence and height of the spikes. The results show that the garbage collection in the JVM, in particular the "stop-the-world" pause, has a strong correlation with the consistency spikes. Furthermore, a method that can virtually eliminate the spikes is proposed. This method entails delaying read operations artificially at server side immediately after garbage collection. In the experiments shown in this chapter, more than 98% of consistency anomalies that exceed a minimum threshold of 5ms are removed with little impact on throughput and latency.

The remainder of the chapter is organized as follows: Section 5.2 presents the background of Cassandra client-side consistency settings, the tuning knob of consistency-availability trade-offs in the system; Section 5.3 shows the hypothesis of how the garbage collection STW in JVM can cause the consistency spikes; the experiments to test the hypothesis and the proposed spike elimination method are detailed in Section 5.4.

5.2 Cassandra Consistency Levels

Cassandra is an open-source distributed storage system that supports tunable consistency among the consistency-availability trade-offs. Users can choose the client-side consistency level for the operations, according to their requirement. This section will briefly explain how Cassandra reads and writes data, and details some consistency levels supported by Cassandra that are related to this study. The descriptions are based on the Cassandra official online documentation [1], but are simplified for presentation. For example, this section only uses the general QUORUM level over other quorum-based levels (e.g., EACH_QUORUM, LOCAL_QUORUM) because this study only focuses on single data center cluster deployment.

In Cassandra, each logical partition of the system is composed of a group of replicas, referred to as *replication group*. The size of the replication group is called *replication factor*. A client can send a request to any server host, and the server will be the coordinator of the operation, forwarding requests to the replica servers accordingly. A replica server *acknowledges* a write request to the coordinator only after the write operation has been appended to the *commit log* on disk and the update is stored in a memory structure

called *memtable* in that server. The following lists some of the consistency levels for write operations supported in Cassandra:

- **ALL.** The operation succeeds when all replicas acknowledge the write. This provides the highest consistency and lowest availability level.
- **QUORUM.** The operation succeeds when a quorum of replicas acknowledge the write. This provides strong consistency in terms of Brewer's CAP principle.
- **ONE.** The operation succeeds when at least one replica acknowledges the write. This provides weak consistency.
- **ANY.** The operation succeeds when at least one node (not necessarily belonging to the replication group) acknowledges the write. When all replicas of a partition are unavailable, ANY allows a write to be temporarily stored on another server and applied when that partition has healed. This mechanism is called *hinted hand-off* in Cassandra, and such a write temporarily stored on another server is called a "hinted write". This level provides the lowest consistency and highest availability.

For read operations, Cassandra has defined similar consistency levels as follows:

- **ALL.** The operation succeeds when all replicas have responded, otherwise the read operation fails. This provides the highest consistency and lowest availability level.
- **QUORUM.** The operation succeeds when a quorum of replicas have responded. This provides strong consistency.
- **ONE.** The operation succeeds when the closest replica has responded. The distance between nodes is determined by the *snitch* setting, which informs the Cassandra about the network topology. This provides the highest availability, but may give rise to stale values.

5.3 Hypothesis

The experiments in [77] used the consistency level "ONE" for both read and write operations. This section uses this consistency level setting for eventual consistency and presents the hypothesis that GC STW causes the consistency spikes, as shown in Figure 1.1 in Chapter 1. The following discussion is under the assumption that the clocks are tightly synchronized

within the cluster, e.g., using network time protocol (NTP). In private clusters, the clock offset under clock synchronization protocols is very small (e.g., offset to the local clock stratum is less than one millisecond), thus the clock offset has little impact on the consistency spikes, some of which are more than 200 milliseconds in Figure 1.1.

Recall that a value returned by a read is stale if that value is not the most recent value that was written to Cassandra, meaning that a subset of the replicas did not receive the last updated value in time for the read to see it. For example, a stale read happens when an acknowledged write operation has succeeded on replica A and not on replica B, then replica B serves a read request using the value it has which is stale compared to the one on A. The staleness of the read operations should be less than the time difference between the times at which the two replicas apply the write. The hypothesis is that this time difference among replicas is enlarged when some replica is in the "stop-the-world" phase. When the JVM of some replica experiences the GC STW pause, all application threads are stopped and that delays the processing of updates at this replica, while the same updates may be applied successfully at other replicas. After application threads resume from STW, any reads from this replica return a stale value until the backlogged updates are applied. Depending on how long it takes to clear the backlog, reads may in theory return values that are stale by more than the duration of the STW pause time at a replica, but this case is rarely observed in the experiments.

Figure 5.1 shows a hypothetical example of the garbage collection "stop-the-world" pause causing a severely stale read. There are three replicas with value 0 initialized by operation w(0). On applying a following write operation W(1), replica 2 is a little behind replica 1 in time, because an eventually consistent system allows asynchronous updates. Thus, there is a small window for an inconsistent read, such as the read R(0) happening on replica 2. In case of garbage collection, as shown in replica 3, STW pauses all read and write threads, resulting in a window of inconsistency which could be as large as the STW duration.

5.4 Experiments

5.4.1 Consistency metric

The experiments presented herein use the Γ (gamma) metric for consistency [48], which is similar theoretically to the metric used in [77] but in practice exhibits less noise in environments where clock offset across hosts exceeds operation latencies. The Γ metric

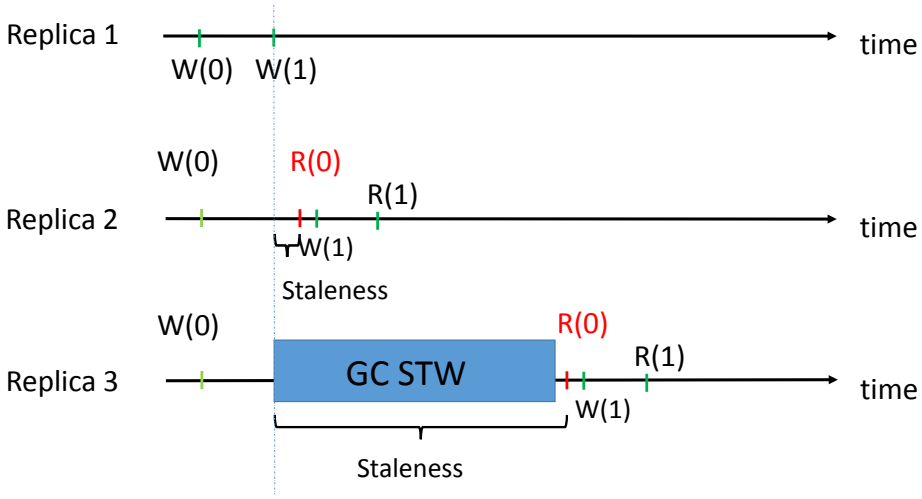


Figure 5.1: Illustration of how STW causes severe staleness.

quantifies how badly the consistency observed by clients deviates from linearizability [53], which states (informally speaking) that each operation appears to take effect instantaneously at some point between its start and finish times as measured from the perspective of the client who applied the operation. The experiments in this chapter measure a fine-grained form of the Γ metric called the *per-value* Γ *score*, which captures deviations from linearizability associated with a collection of operations that access the same key and read or write the same value. Positive Γ scores indicate consistency anomalies, which can be interpreted as stale reads or as write operations that appear to take effect in a non-linearizable order. To plot a time series graph, each Γ score is plotted against a point in time, defined relative to the beginning of an experiment, which indicates approximately when the corresponding consistency anomaly occurred. The time values are approximate since anomalies involve the interaction of multiple storage operations that may start and finish at different times.

5.4.2 Hardware and software environment

The experiments are conducted in a private cluster of 11 Intel Xeon E5450 8-core commodity machines with 8GB RAM, connected by Gigabit Ethernet. The hosts use a 64-bit Linux kernel version 2.6.18 and provide Oracle Java 1.7.0u71. Cassandra version 2.0.9 is installed and configured using default parameters except where noted otherwise. A modified version of YCSB [28] 0.1.4 is used, similarly to [77], to collect logs of operations from which the consistency metric is computed. One host is used as a coordinator to monitor the experiment

and collect logs, five hosts run Cassandra, and up to five other hosts run multi-threaded YCSB clients.

The Oracle HotSpot JVM provides garbage collectors that pause the application threads to evacuate the garbage objects. This pause time is also called the "stop-the-world" (STW) time. Even concurrent garbage collectors such as Concurrent Mark Sweep (CMS) include a STW pause in the mark and remark steps. To correlate the inconsistency spikes with GC pause time, the experiments collect the STW start and finish timestamps by parsing the garbage collection logs of the JVMs running Cassandra.

The YCSB workload is run against Cassandra for 120 seconds. Unless specified otherwise, the workload settings are as follows: hotspot distribution with 80% of the load on 20% of the keys, key space size of 500, 32 client threads per YCSB process, and 80%/20% read/write operation mix. A skewed distribution is used to maximize the likelihood of observing consistency anomalies, as in [48, 77]. Cassandra is configured with a replication factor of 3 and consistency level "ONE" is used for both reads and writes, which means that the client waits for only one replica to return an acknowledgment. For each experiment, the graph presents the result of one run, but each setting is repeated five times to confirm that the pattern observed is reproducible.

5.4.3 Inconsistency spikes versus STW pause

Figure 5.2 shows the result of reproducing the staleness time series experiment of Rahman et al. (see Figure 3 in [77]) using the Γ metric, where the time and duration of the GC STW pause are shown as vertical dotted lines in the same graph. The position of these lines on the x-axis indicates the time of an STW pause, and the height of these lines on the y-axis indicates the duration of the pause. Both GC STW pause times and Γ metric values are aggregate results from all hosts. For the sake of clarity, STW times less than 5 ms are not plotted. The spike pattern in Figure 5.2 at around 40 seconds is very similar to Figure 3 in [77] at around 55 seconds.

Informally speaking, the spikes demonstrate a strong correlation with the GC pause. In terms of experiment time the spikes align precisely with the STW pauses indicated by the dotted lines perpendicular to the x-axis. Furthermore, the height of the spikes corresponds closely to the length of the GC pause, which is indicated by the height of the dotted lines. In other words, nearly all consistency violations happen at a time near a GC STW pause, and most of the observed Γ scores, which quantify the severity of a consistency violation, are less than the duration of the pause.

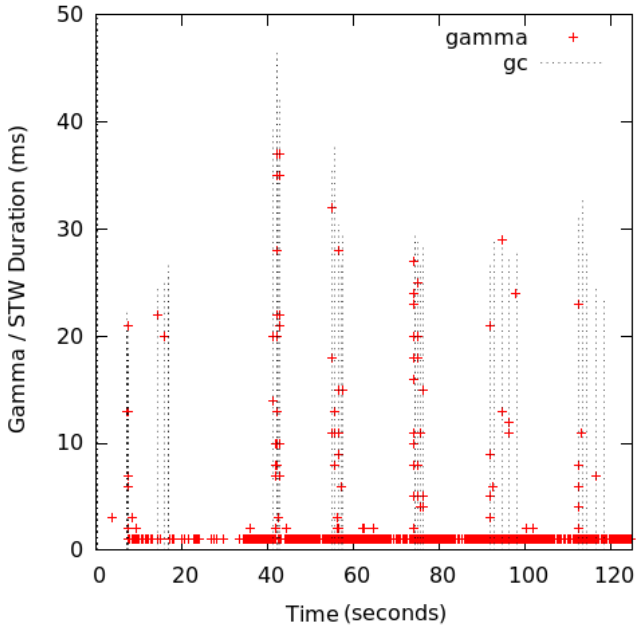
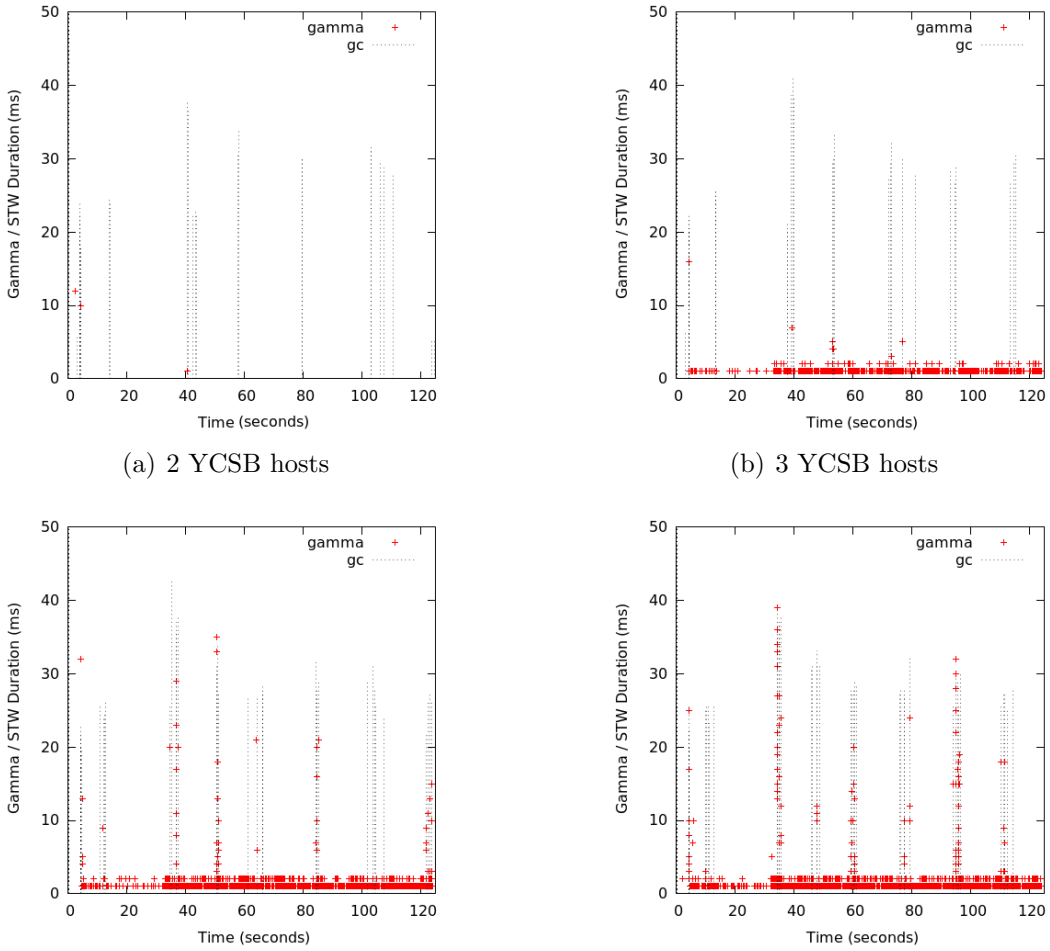


Figure 5.2: Time series of Γ score and GC pause time with five YCSB hosts and replication factor three.

To explore the internal cause of the inconsistency spikes, the experiments with a fixed number of Cassandra hosts but a various number of YCSB hosts are conducted. In this experiment the replication factor is set to 5 to allow each Cassandra host to hold one replica. Figure 5.3 shows the results using 2 to 5 YCSB hosts and 5 Cassandra hosts. As the number of YCSB hosts is varied, each YCSB host has similar throughput, and the aggregate throughput is proportional to the number of YCSB hosts. It is observed that as the number of YCSB hosts increases, the consistency violations become more severe but generally do not exceed the length of the STW pause; this observation will be exploited later on in Section 5.4.4. Furthermore, the number of spikes and the number of positive Γ scores increases more than linearly with the number of YCSB hosts (e.g., compare Figure 5.3 (a) with (c)). Thus, consistency anomalies occur readily as the offered load increases, and may be of concern in practice even when the storage system is operating at less than full throttle.



(a) 2 YCSB hosts

(b) 3 YCSB hosts

(c) 4 YCSB hosts

(d) 5 YCSB hosts

Figure 5.3: Time series of Γ score and GC pause time with replication factor five.

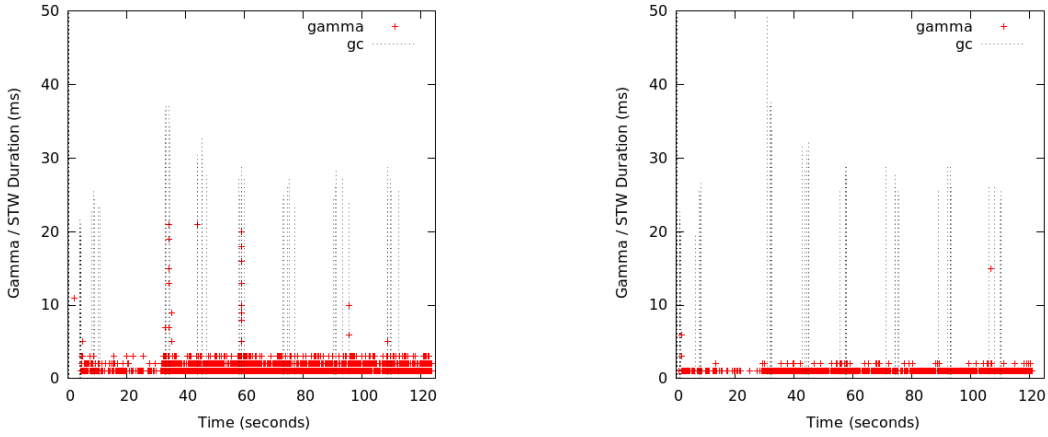
5.4.4 Smoothing the inconsistency spikes

This subsection investigates the technique of smoothing out the inconsistency spikes by delaying reads artificially during a short interval of time following a GC STW pause, which is referred to as the *delay period*. The delay period begins as soon as the GC STW pause is detected, and lasts for a duration that depends on the method used to detect the pause. Operation delays are governed by the following policy: inside the delay period, read operations are stalled until the delay period is finished, whereas write operations follow

the usual execution path; outside the delay period reads and writes both follow the usual execution path.

Two concurrent garbage collectors are tested in the experiments: ConcurrentMarkSweep (CMS), the default GC, and Garbage First (G1), a newer GC provided by the HotSpot JVM. The results for both garbage collectors are similar, and so only the results for CMS are presented.

Two mechanisms to detect the GC STW pause are used: *GC-notification* and *Free-heapsize*. Since Java 7 update 4, an API called *GarbageCollectionNotificationInfo* is available for receiving notifications from the GarbageCollectorMXBean. It can provide the start time and duration of the GC, which are used to define the delay period. Once a GC occurs, the delay period is activated in the notification handler for the GC duration specified by the management bean. This method is called as *GC-notification*. The *Free-heapsize* method instead traces free heap size in the read operation routine. Once a large free heap size increase ($\geq$100MB in the setting) is detected, it is considered to be a sign of garbage collection occurring. In that case, the delay period is activated for a fixed period of 50ms—an empirically determined upper bound on the length of the STW pause in the experiments. As noted earlier in Section 5.4.3 the length of the STW pause is, in turn, an approximate upper bound on the Γ scores observed during inconsistency spikes.



(a) GC-notification method

(b) Free-heapsize method

Figure 5.4: Smoothing of inconsistency spikes by delaying reads artificially after a GC STW pause.

Figure 5.4 (a) shows the result of using the GC-notification method with a replication factor of five and five YCSB hosts. In terms of the number of positive Γ scores and the height

	#violations ($\Gamma > 5$ms)	Aggregate Throughput (ops/s) ± std err	Average Latency (ms) ± std err	Max Latency (ms) ± std err	95%-ile Latency (ms) ± std err
No delay	133	8604 ± 10	9.23 ± 0.02	324 ± 11	25.0 ± 0
GC notification	31	8584 ± 21	9.33 ± 0.03	311 ± 4	25.6 ± 0.2
Free heapsize	2	8586 ± 15	9.38 ± 0.03	322 ± 13	25.8 ± 0.2

Table 5.1: Comparison of consistency violation, throughput and read latency under read delay in Apache Cassandra.

of the spikes, the consistency anomalies are less severe than in Figure 5.3 (d), which uses the same system and workload parameters. However, there are still three spikes in excess of 20ms in height. Thus, the smoothing is incomplete, possibly because the GarbageCollection notification handler is not guaranteed to execute in a timely fashion. In other words, the execution of the notification handler that activates the delay period may lag behind the end of the STW pause.

Figure 5.4 (b) shows the results using the Free-heapsize method. It demonstrates that the spikes are nearly removed and 98.5% Γ scores more than 5ms are also removed. With the exception of one Γ score of around 6ms at the beginning of the experiment, and one Γ score of around 15ms at the 108th second, all the remaining points are under 5ms.

Table 5.1 shows the throughput and read latency influence of the artificial read delays. GC notification has no significant drop in throughput, 1% increase in average read latency, no significant increase in max read latency, and 0.6ms increase in 95%-ile read latency. Free heapsize has no significant drop in throughput, 1.6% increase in average read latency, no significant increase in max read latency, and 0.8ms increase in 95%-ile read latency. Thus, the results show that the spikes can be virtually eliminated with very little overhead in terms of throughput and latency.

5.5 Summary

In this chapter, the inconsistency spikes observed in [77] were reproduced in experiments. It was experimentally shown that the stop-the-world pause during garbage collection is the cause of these spikes. Specifically, the results showed a strong correlation between the GC pauses and inconsistency spikes in terms of both the time of occurrence and the GC duration vs. spike magnitude. The explanation of GC STW causing updates to be applied

at different times at various replicas was formed. Furthermore, two methods of delaying read operations artificially to smooth out the spikes were demonstrated: GC-notification and Free-heapsize. Both methods incurred only a slight overhead in throughput and read latency, and successfully removed up to 98% of the spikes.

Chapter 6

Related Work

The chapter discusses the related work in the area of scalable distributed databases, with an emphasis on systems that favor consistency.

Distributed transactions. Distributed transactions have been studied since the 1980s when transaction processing began to involve multiple partitions [21]. Distributed transactions are desirable and attract increasing attention [14, 20, 34, 52, 66, 70, 71, 85, 101, 104] in both industry and research, because data generation and transaction processing exceed the capacity of a single partition in many use cases. In addition, it is challenging to partition data so that transactions can only access a single partition [56, 74]. Prior distributed storage systems mostly either pay a performance penalty for serializable distributed transaction or sacrifice serializability in favor of weaker isolation models. Sinfonia [9] provides serializable minitransactions, which internally use lock-based concurrency control and a two-phase transaction commitment protocol. Spanner [29], which accommodates a broader class of transactions, uses a similar mechanism combined with optimistic concurrency control, as well as data versioning to avoid locks for read-only transactions. In contrast, VoltDB [87] executes transactions serially in each partition, which avoids the need for concurrency control and enables high throughput for single-partition transactions but forces centralized coordination for multi-partition transactions. MDCC [59] and RAMP [14] use commitment protocols in which the number of network round trips depends on the contention encountered. MDCC provides read committed isolation by default and RAMP provides read atomicity, which is also weaker than serializability. None of the above systems allow distributed write-only transactions to be committed in amortized one round trip in the presence of write-write conflicts.

Concurrency control for serializability. Moreover, serializable distributed trans-

actions are inherently costly [12] as the transaction coordination over multiple partitions across the network is coupled with the concurrency control mechanisms, which may force a transaction to wait for others to complete. There are many efforts in the research community to support serializable distributed transactions. Many of these use variants of traditional concurrency control mechanisms that originally work well in single-partition transactions: for example, Spanner [29] and Sinfonia [9] use two-phase locking, whereas Centiman [34], Rococo [70], Tapir [104], Hyder [22], and Hekaton [33] implement OCC. However, these concurrency control mechanisms do not scale well in the presence of conflicts, in particular in update-intensive workloads. A recent evaluation [52] shows that for these conventional concurrency control mechanisms (excluding deterministic scheduling in Calvin), the performance of distributed transactions on a cluster only slightly exceeds that of a single machine under a high contention read-write workload. That means the potential capacity of the cluster is underutilized because of the concurrency control performance bottleneck.

Some recent systems propose architectures and concurrency control mechanisms that are designed for distributed transactions. For example, some of them address distributed transactions through deterministic scheduling [37, 38, 80, 87, 92, 93], by appending to a log for atomic commitment [19, 22, 46], or by re-ordering conflicting transaction executions [70]. Calvin [80, 92, 93] processes transactions in a deterministic order, which makes it possible to process write-only transactions in one round trip. However, as shown in the experiments of Chapter 4, Calvin suffers from the latency overheads related to the scheduling phase, and its open source implementation cannot abort transactions using a second round even if some partition is overloaded or failed [80]. Faleiro et al. extends Calvin using a placeholder approach [37, 38], but their systems cannot execute conflicting transactions in parallel using key-level concurrency control. Hyder [19, 22] shares some similarities with ALOHA-DB in terms of achieving atomic multi-write with low overhead without resorting to 2PC, using multi-versioning instead of locking, and separating transaction execution into multiple stages (i.e., recording the transaction first and then executing it). However, Hyder achieves atomic multi-key writing by atomically writing a log entry in Corfu [16], which uses a centralized *sequencer* [1] to guarantee a total order on the log entries. This sequencer can limit the peak write throughput of Hyder to only sub-million transactions per second based on the current generation of hardware [16]. ALOHA-DB instead orders transactions using the timestamps assigned by distributed front-end (FE) servers. Furthermore, to read the values of some keys, Hyder must "roll forward" all the sequential log entries, even if some of the entries do not affect the keys of interest, whereas ALOHA-DB only needs to compute the functors that are related to the keys of interest. Also, Hyder transactions are prone to aborting during the "melding" phase under high contention, whereas ALOHA-DB never aborts transactions due

[1] The sequencer in Corfu and Hyder has different functionality from that in Calvin.

to conflicts. HANA SOE [46] proposes another log-structured scale-out database similar to Hyder, but uses a transaction manager for concurrency control instead of the intention-meld mechanism. ROCOCO [70] is a concurrency control mechanism for distributed transactions that uses a two-round protocol to detect conflicts and re-order transaction execution. The ALOHA-DB system determines the transaction order by the timestamps generated at the start of transactions, and no conflict detection between transactions is needed. ALOHA-DB has superior performance compared to published results for ROCOCO (around 55000 transactions per second across 100 single core machines) [70].

Using epochs to batch operations. Previous works use epochs to structure transaction processing in various ways [58, 72, 95, 76]. Silo [95] and its more scalable relative FOEDUS [58] use epochs to ensure serializability on recovery from failures, to facilitate garbage collection, and to provide read-only snapshots. Phase reconciliation [72] repeatedly cycles *split phases*, epochs that only process commutative operations, and *joined phases*, epochs that process other operations. Phase reconciliation resembles ECC on the first impression, but isolation of reads from writes in our system is orthogonal to the isolation of commutative and non-commutative operations in their system. For example, two writes on the same key do not commute and thus cannot be processed concurrently in phase reconciliation, but can be processed in parallel in ALOHA-KV and ALOHA-DB. Additionally, these other systems target single machine transactions, whereas ALOHA-KV and ALOHA-DB targets distributed transactions.

A recent paper on scale replay recovery logs on replica [76], referred to as *replay replication*, shares some similarity with ALOHA-DB by using the techniques of epochs, deterministic execution, multi-versioning, as well as a placeholder approach, but targets a different purpose (replay recovery log). Functor-enabled ECC itself is a concurrency control mechanism, while the replay replication paper addresses log replay in primary-backup replication, which requires another concurrency control mechanism (on the primary server) to decide transaction ordering and to create epochs for the backup server. Furthermore, the replay replication paper proposes a technique designed for multi-core databases that do not support distributed transactions, while ALOHA-DB is designed for distributed transactions. Last but not the least, the placeholder in replay replication is only an empty value, and it only achieves transaction-level parallelism; functors are placeholders and methods to get the final values, and functors allow key-level parallelism in transaction processing.

Read-only and write-only transactions. Efficient read-only and write-only transaction protocols are proposed in prior works, such as Eiger [65], Walter[85], and G-store [30], and can be used in batch processing systems [26, 54]. For these transaction types, ALOHA-KV also benefits from batching, and achieves throughput-optimized performance even under high contention. However, Eiger [65] only supports causal consistency, and Walter [85]

provides parallel snapshot isolation, while ALOHA-KV provides serializable isolation. G-store [30] only supports the multi-key access where the keys are in non-overlapping groups. Thus, G-store can not efficiently handle the general case that two concurrent write-only transactions contend for some common keys.

Quantifying eventual consistency. Recent research has been working on quantifying eventual consistency, informally the staleness of data returned by reads relative to the latest data that has been written. Wada [98] et al. evaluated the staleness of Amazon's SimpleDB using end-user stress-testing. Bermbach and Tai quantified various forms of non-serializable behavior on S3 [18]. Zellag and Kemme [102, 103] analyzed consistency by building dependency graphs similar to graphs that are used to characterize serializability. Bailis et al. [15] developed a model to predict the expected staleness bound, called probabilistically bounded staleness (PBS). Golab et al. developed the Δ [47] (delta) and Γ [48] (gamma) metrics to measure how far the operation history deviates from linearizability. The measured staleness metrics can be used in various applications, such as consistency analysis tools for storage systems [40] and systems that tune the consistency-latency tradeoff [25, 78].

Chapter 7

Conclusion and Future Work

7.1 Conclusions

This thesis addresses the fundamental question of building distributed databases: how to improve performance and consistency under conflicts. Conflict mitigation, a new technique that targets overall scalability and consistency in conflict resolution, is proposed. This thesis includes research on high performance serializable distributed transactions and an investigation of consistency spikes in eventually consistent systems. In particular, it has the following contributions.

A concurrency control mechanism for distributed transactions. The main contribution of this thesis is a suite of high performance distributed transaction protocols. At the center of these protocols, a novel concurrency control mechanism for atomic read-only write-only transactions is proposed, which is called epoch-based concurrency control (ECC). ECC avoids read-write conflicts among transactions by partitioning transaction execution into disjoint read and write epochs, and mitigates write-write conflicts by storing multiple versions of key-value pairs. Thus, concurrent writes can be processed in parallel with low overhead, even when their write sets overlap. Based on ECC, the high performance distributed databases ALOHA-KV and ALOHA-DB are built, both of which guarantee serializability.

An efficient atomic commitment protocol. Using ECC as the central building block, this thesis describes a new atomic commitment protocol, which requires amortized one round trip to commit a transaction in the absence of failures irrespective of contention. In comparison, the widely used two-phase commit protocol requires two round trips even if

there is no failure. Different from deterministic database Calvin, which submits transactions in one round trip and enforces the transaction execution by deterministic order, the commitment protocol proposed in this thesis still has the ability to abort the transaction using a second round of messages.

An asynchronous processing paradigm for serializable transactions. Functors, proposed in this thesis, extend the ECC to support high performance read-write distributed transactions. The functor-enabled ECC uses write epochs to record functors, which are objects that represent how to evaluate the corresponding versions of values, and process the functors either asynchronously or at read time, once all versions on which the functor depends are settled. Functors allow a finer concurrency control level in transaction processing, while enabling serializability.

An investigation of consistency spikes in an eventually consistent system. Consistency spikes in a popular eventually consistent system are reproduced in the thesis. A strong correlation between garbage collection "stop-the-world" pauses and consistency spikes is demonstrated. Furthermore, the explanation of the correlation is formulated theoretically and tested experimentally. In addition, this research proposes a solution to smooth out the consistency spikes by artificially delaying read operations immediately after garbage collection. The experimental evaluation of this technique shows that the spikes can be virtually eliminated with little impact on throughput and latency.

7.2 Future Research Directions

A few promising directions for future work can be motivated based on this thesis. This subsection presents the future work based on the directions.

Improving the performance of ECC

1. *Epoch duration self-tuning.* ECC introduces an interesting tuning knob for adaptivity with respect to workload variations, such as changes between read-heavy and write-heavy workloads, or large versus small transactions. However, an interesting future work is an intelligent self-tuning mechanism that will adjust the epoch durations dynamically in response to the workload. For example, (1) what metrics can be used to represent the workload; (2) how to decide the new epoch duration based on the metrics; (3) how the system adapts to the new epoch duration?

2. *Reducing epoch switch time using low latency network.* Reducing epoch switch time will further reduce the overhead of ECC, which has positive impacts on both the

throughput and the latency of the system. It is an interesting research direction to speed up epoch switch time using new network hardware, such as remote direct memory access (RDMA), which currently achieves much smaller latency than TPC/IP over Ethernet. In addition, implementing part of the network stack in hardware using an FPGA is another possible solution for reducing the computational overhead of epoch switching.

3. *Exploring ECC in multi-core machines.* Inspired by recent developments in transaction processing on a single many-core machine [95, 100], further research may explore whether the benefits of ECC demonstrated in a distributed message passing system can be realized in shared memory as well.

Facilitating functor usage in transaction processing

1. *Automating functor transformation.* Even though Section 4.3.3 described a general but naive method to transform the transaction stored procedure to functor handlers, the current implementation of ALOHA-DB still uses the manually created functors because no automated functor transformation program exists yet. There are several interesting research questions regarding creating such a program. First, what do the input and output for such an automated transformation look like? I.e., how to represent a stored procedure and a functor handler in the course of the transformation? Second, is there a transformation method that can generate more efficient functors for transaction processing than the one described in Section 4.3.3? Third, how to automatically evaluate which functor is more efficient if there exist multiple ways of performing the transformation? For example, if some benchmark framework is available, how to automatically decide what testing workloads will be applied to the candidate functors?

2. *Processing user-defined functors.* In Chapter 4, functors are computed by stored procedures, which are installed in the server side before transaction processing, and the clients send the transactions to servers using the procedure IDs (f-types) to indicate which procedures to call. Is it possible to eliminate the restriction that the functors are computed by stored procedures and the clients must have prior knowledge of the procedure IDs? This future work focuses on techniques that allow clients to send the user-defined functors created at run time. One idea is that the functors are written by a newly designed language (e.g., scripts or a SQL-like language) and this language can be interpreted and be executed by a library running on the server side.

3. *Building ALOHA-DB on top of storage class memory.* The forthcoming generation of storage media, such as the *storage class memory (SCM)*, may raise more interesting design questions regarding providing data durability in ALOHA-DB. For example, the logging and checkpointing mechanism may be not necessary with SCM, because the data in SCM will not be lost due to crashing. However, processes may crash and leave an incomplete state in the SCM, which needs to be resolved in the recovery program.

Supporting cross-datacenter transactions

This thesis currently focuses on clusters of servers that are in a single datacenter. However, these techniques face challenges in the cross-datacenter environment, where the network latency is much larger than in the single datacenter case and the failure detection and resolution are much slower. The following are some questions that need to be addressed to support cross-datacenter transactions.

1. *New epoch switch mechanism.* In a single datacenter scenario, epoch switch time is much smaller than the epoch duration (based on the experiments presented in previous chapters). But the situation may be different in cross-datacenter scenarios, because network latency for one round trip can be less than 1 millisecond in a single datacenter but may be greater than 100 milliseconds in geo-distributed scenarios. The epoch switch mechanism needs to be re-designed to accommodate large network latency.

2. *Distributed epoch manager.* Other than a centralized epoch manager (EM) used in experiments of this thesis, the cross-datacenter scenario may need a distributed epoch manager. Otherwise, the centralized EM may be a performance bottleneck for serving servers in remote data centers. In addition, the clock offset among servers within a single datacenter is smaller than that in cross-datacenter environment, as the network latency is much larger in the latter case. The epoch manager design needs to accommodate the large the clock offset scenario.

3. *Quorum-based geo-replication.* When replicas are in different data centers, the replication protocol for synchronizing cross-datacenter replicas needs to deal with large network latencies and large latency variations. One direction of future work is to explore the quorum-based replication in ALOHA-DB for cross-datacenter transaction support.

Further smoothing consistency spikes in eventually consistent systems

1. *Combination of GC-notification and Free-heapsize.* In servers with larger main memories, the GC pause time may on occasion be much longer than in the experiment in this thesis, which could require an excessively long fixed delay period duration in Free-heapsize. On the other hand, the notification lag of the GC notification method is usually quite small. Combining the two strategies could provide both agile reaction and a precise calculation of the delay period duration.

2. *Tuning JVM settings for shorter garbage collection pause time.* As the consistency spikes result from the GC pause, ordinary GC tuning techniques may reduce the spikes by shortening the STW pause time. For example, one could tune the size of the young generation in heap. Real-time JVMs such as C4 [89], or (nearly) pauseless garbage collectors such as Shenandoah [5], may also be helpful.

3. *Controlling internal activities in the storage system.* Consistency anomalies could be made more predictable to avoid inconsistency at inconvenient times, by means of explicitly invoking GC, or controlling internal activities that may lead to GC. For example, in Cassandra such internal activities include the *flush*, which flushes in-memory data from the *memtable* to an *SSTable* file on disk, and may be accompanied by a substantial GC since a large chunk of memory becomes available for recycling.

References

[1] Apache Cassandra documentation. http://docs.datastax.com/en/cassandra/2.0/. Accessed: 2015-05-15.

[2] Cassandra. http://cassandra.apache.org/ Accessed: 2015-08-21.

[3] HBase. http://hbase.apache.org/ Accessed: 2015-08-21.

[4] Project Voldemort. http://www.project-voldemort.com/voldemort/ Accessed: 2015-08-21.

[5] Shenandoah. http://icedtea.classpath.org/wiki/Shenandoah Accessed: 2015-08-21.

[6] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, February 2012.

[7] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, Cambridge, MA, USA, March 1999. Also as Technical Report MIT/LCS/TR-786.

[8] Marcos K. Aguilera, Joshua B. Leners, Ramakrishna Kotla, and Michael Walfish. Yesquel: Scalable SQL Storage for Web Applications. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, pages 40:1–40:4, New York, NY, USA, 2015. ACM.

[9] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, November 2009.

[10] AWS. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2017. Accessed: 2017-11-21.

[11] Peter Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, University of California, Berkeley, CA, USA, 2015.

[12] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[13] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, Not CAP: Towards Highly Available Transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association.

[14] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 27–38, New York, NY, USA, 2014. ACM.

[15] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014.

[16] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, December 2013.

[17] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[18] David Bermbach and Stefan Tai. Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3's Consistency Behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[19] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1295–1309, New York, NY, USA, 2015. ACM.

[20] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[21] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[22] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A transactional record manager for shared flash. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 9–20, 2011.

[23] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging NVM: A survey on architectural integration and research challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2):14:1–14:32, November 2017.

[24] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 7, New York, NY, USA, 2000. ACM.

[25] Shankha Chatterjee and Wojciech Golab. Self-tuning eventually-consistent data stores. In Paul Spirakis and Philippas Tsigas, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 78–92, Cham, 2017. Springer International Publishing.

[26] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 169–182, New York, NY, USA, 2012. ACM.

[27] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[28] 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. ACM.

[29] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter

Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[30] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 163–174, New York, NY, USA, 2010. ACM.

[31] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726. VLDB Endowment, 2006.

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

[33] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[34] Bailu Ding, Lucja Kot, Alan Demers, and Johannes Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 262–275, New York, NY, USA, 2015. ACM.

[35] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[36] Facebook. fbthrift. https://github.com/facebook/fbthrift, 2015. Accessed: 2017-11-21.

[37] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, January 2017.

[38] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 15–26, New York, NY, USA, 2014. ACM.

[39] Hua Fan. High performance multi-partition transaction. In *Proceedings of the VLDB 2015 PhD Workshop co-located with the 41rd International Conference on Very Large Databases (VLDB 2015), Hawaii, USA, August 31, 2015.*

[40] Hua Fan, Shankha Chatterjee, and Wojciech M. Golab. Watca: The waterloo consistency analyzer. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1398–1401, 2016.

[41] Hua Fan, Wojciech Golab, and Charles B. Morrey, III. ALOHA-KV: High performance read-only and write-only distributed transactions. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 561–572, New York, NY, USA, 2017. ACM.

[42] Hua Fan, Aditya Ramaraju, Marlon McKenzie, Wojciech Golab, and Bernard Wong. Understanding the causes of consistency anomalies in apache cassandra. *Proc. VLDB Endow.*, 8(7):810–813, February 2015.

[43] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM.

[44] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[45] 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.

[46] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, August 2015.

[47] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS*

Symposium on Principles of Distributed Computing, PODC '11, pages 197–206, New York, NY, USA, 2011. ACM.

[48] Wojciech Golab, Muntasir Raihan Rahman, Alvin Au Young, Kimberly Keeton, Jay J. Wylie, and Indranil Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 28:1–28:2, New York, NY, USA, 2013. ACM.

[49] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[50] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[51] Rajiv Gupta and Charles R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *Int. J. Parallel Program.*, 18(3):161–180, June 1990.

[52] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, January 2017.

[53] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[54] Charles Johnson, Kimberly Keeton, Charles B. Morrey, Craig A. N. Soules, Alistair Veitch, Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer, Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro. From research to practice: Experiences engineering a production metadata database for a scale out file system. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 191–198, Berkeley, CA, USA, 2014. USENIX Association.

[55] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 603–614, New York, NY, USA, 2010. ACM.

[56] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, August 2008.

[57] Rajesh Kumar Karmani, Nicholas Chen, Bor-Yiing Su, Amin Shali, and Ralph Johnson. Barrier synchronization pattern. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, 2009.

[58] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM.

[59] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[60] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[61] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[62] Leslie Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, Jun 1986.

[63] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, June 1988.

[64] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[65] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[66] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 135–150, Berkeley, CA, USA, 2016. USENIX Association.

[67] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[68] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[69] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[70] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Berkeley, CA, USA, 2014. USENIX Association.

[71] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 517–532, Berkeley, CA, USA, 2016. USENIX Association.

[72] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 511–524, Berkeley, CA, USA, 2014. USENIX Association.

[73] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[74] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[75] Francisco Perez-Sorrosal, Marta Patiño Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. Elastic SI-Cache: Consistent and scalable caching in multi-tier architectures. *The VLDB Journal*, 20(6):841–865, December 2011.

[76] Dai Qin, Angela Demke Brown, and Ashvin Goel. Scalable replay-based replication for fast databases. *Proc. VLDB Endow.*, 10(13):2025–2036, September 2017.

[77] Muntasir Raihan Rahman, Wojciech Golab, Alvin AuYoung, Kimberly Keeton, and Jay J. Wylie. Toward a principled framework for benchmarking consistency. In *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, Berkeley, CA, USA, 2012. USENIX Association.

[78] Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 11(4):20:1–20:36, January 2017.

[79] Kun Ren. Calvin. https://github.com/yaledb/calvin, 2015. Accessed: 2017-11-21.

[80] Kun Ren, Alexander Thomson, and Daniel J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 7(10):821–832, June 2014.

[81] Lawrence G. Roberts. ALOHA packet system with and without slots and capture. *SIGCOMM Comput. Commun. Rev.*, 5(2):28–42, April 1975.

[82] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1311–1326, New York, NY, USA, 2015. ACM.

[83] 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.

[84] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng.*, 9(3):219–228, May 1983.

[85] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on*

Operating Systems Principles, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[86] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[87] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[88] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, November 2014.

[89] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *SIGPLAN Not.*, 46(11):79–88, June 2011.

[90] Douglas B. Terry, Karin Petersen, Mike J. Spreitzer, and Marvin M. Theimer. The case for non-transparent replication: Examples from bayou. *In IEEE Data Engineering*, 21:12–20, 1998.

[91] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[92] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1-2):70–80, September 2010.

[93] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[94] Transaction Processing Performance Council (TPC). TPC Benchmark C standard Sepcification Revisiosn 5.11. http://www.tpc.org/tpcc, 2010. Accessed: 2017-11-21.

[95] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of SOSP '13*, pages 18–32, New York, NY, USA, 2013. ACM.

[96] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[97] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. TAO: How facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 791–792, New York, NY, USA, 2012. ACM.

[98] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, Online Proceedings*, pages 134–143, 2011.

[99] Jenq-Shyan Yang and Chung-Ta King. Designing tree-based barrier synchronization on 2d mesh networks. *IEEE Trans. Parallel Distrib. Syst.*, 9(6):526–534, June 1998.

[100] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.

[101] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[102] Kamal Zellag and Bettina Kemme. How consistent is your cloud application? In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 6:1–6:14, New York, NY, USA, 2012. ACM.

[103] Kamal Zellag and Bettina Kemme. Consistency anomalies in multi-tier architectures: Automatic detection and prevention. *The VLDB Journal*, 23(1):147–172, February 2014.

[104] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In

Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, pages 263–278, New York, NY, USA, 2015. ACM.