Fast Transactions in Distributed and Highly Available Databases

by

Yi Lu

B.E., Harbin Institute of Technology (2013)
M.Phil., The Chinese University of Hong Kong (2015)
S.M., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering
and Computer Science
July 15, 2020

Certified by .....
Samuel R. Madden
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Fast Transactions in Distributed and Highly Available Databases

by

Yi Lu

Submitted to the Department of Electrical Engineering
and Computer Science
on July 15, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Many modern data-oriented applications are built on top of distributed OLTP databases for both scalability and high availability. However, when running transactions that span several partitions of the database, significant performance degradation is observed in existing distributed OLTP databases. In this thesis, we develop three systems — (1) STAR, (2) COCO, and (3) Aria — to address the inefficiency and limitations of existing distributed OLTP databases while using different mechanisms and bearing various tradeoffs. STAR eliminates two-phase commit and network communication through asymmetric replication. COCO eliminates two-phase commit and reduces the cost of replication through epoch-based commit and replication. Aria eliminates two-phase commit and the cost of replication through deterministic execution. Our experiments on two popular benchmarks (YCSB and TPC-C) show that these three systems outperform conventional designs by a large margin. We also characterize the tradeoffs in these systems and the settings in which they are most appropriate.

Thesis Supervisor: Samuel R. Madden
Title: Professor of Electrical Engineering and Computer Science

To my family and friends

Acknowledgments

I still remember that moment when I first arrived at Boston Logan International Airport. I posted a picture to my Instagram and captioned it with "In 2015, I came to the United States with two bags and one dream". It's hard to believe how quickly time flies and my PhD journey has come to an end. As I look back over my life, I realize that it would not be possible to finish my PhD study without the help of many people.

I am extremely grateful to have a great advisor Samuel Madden, who has always been a source of encouragement and inspiration. Sam has given me freedom on research since the first day I came to MIT. With his trust, I have explored many different research areas in my PhD study such as data warehousing and transaction processing. The completion of my thesis would not have been possible without his support.

I would like to extend my sincere thanks to my thesis committee: Tim Kraska and Robert Morris, who gave me invaluable comments, with which I have greatly improved my thesis.

Thanks should also go to my research collaborators: Lei Cao, Franz Franchetti, Alekh Jindal, Elke A. Rundensteiner, Anil Shanbhag, Daniele G. Spampinato, Yizhou Yan, Xiangyao Yu and Jiyuan Zhang with whom I've written and published many papers at top conferences.

I cannot leave MIT without mentioning the members of the MIT database group and data systems group with whom I've had many fruitful discussions over the last five years. Thanks should also go to Dorothy Curtis who is always available to solve all technical issues and Sheila Marian who takes care of everything beyond research.

I also wish to thank my mentors Sudipto Das at Microsoft Research and Wolfram Schulte at Facebook who I worked with during my summer internships. It's them who let me know how the life of a researcher or an engineer looks like in industry.

Special thanks to James Cheng, who was my master thesis advisor at the Chinese University of Hong Kong. He introduced me to the world of database research and

gave me tremendous help when I applied to graduate school.

I am also grateful to have been supported by the Facebook PhD Fellowship from 2018 to 2020.

I would like to thank everyone who helped me in the past years. Thanks to Hongyan Wang and Taosha Wang who helped me settle down in Cambridge. Thanks to Jian Jiang who introduced me to the world of photography. Thanks to Lei Cao who always knows the best restaurants and hiking places. Thanks to Xiong Zhang who did not leave me alone to fight the black bear when we went hiking in Great Smoky Mountains National Park. Thanks should also go to Jiyuan Zhang who is always there and willing to cook delicious food for me.

As a nature lover, it's also enjoyable to hit the road with my friends for vacation. I must also thank Mange Chen, Qi Guo, Tianyi Huang, Kaiyu Li, Liyuan Liu, Zeyuan Shang, Ji Sun, Danruo Wang, Zhongqiu Wang, Xin Xu, Guohong Yang, Xu Yang and Hong Zhou for traveling with me to many stunning national parks and more than thirty states in the United States.

Finally, I thank my parents for giving me their unconditional encouragement and support. They have always told me to aim high. Without their encouragement, I might have never applied and been admitted to MIT. Now, I am finishing my PhD study. I love you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many modern data-oriented applications are built on top of distributed OLTP[1] databases both for scalability and high availability, since the capacity and availability of single-node databases fail to meet many demanding application. In distributed databases, scalability is achieved through data partitioning [22, 91], where each node contains one or more partitions of the whole database. Partitioning-based systems can easily support *single-partition transactions* that run on a single node and require no coordination between nodes. If a workload consists of only such transactions, it trivially parallelizes across multiple nodes. However, distributed databases become more complex when dealing with *distributed transactions* that touch several data partitions across multiple nodes and significant performance degradation as noted below.

In addition, a desirable property of any distributed database is high availability, i.e., when a server fails, the system can mask the failure from end users by replacing the failed server with a standby machine. High availability is typically implemented using data replication, where all writes are handled at the primary replica and are shipped to the backup replicas. However, significant performance degradation is observed in existing distributed OLTP databases when ACID properties and strong consistency between replicas are enforced [5, 22, 74, 96]. For example, strongly consistent databases normally require at least one network round trip to commit a transaction.

[1] Online Transactional Processing

1.1 Research Challenges

In this section, we describe in more details the two major research challenges in distributed and highly available database systems that we address in this thesis.

1.1.1 Distributed Transactions

Distributed databases become more complex when dealing with *distributed transactions* that touch several data partitions across multiple nodes. In particular, many implementations require the use of two-phase commit (2PC) [65] to enforce atomicity and durability, making the effects of committed transactions recorded to persistent storage and survive server failures.

It is well known that 2PC causes significant performance degradation in distributed databases [5, 22, 74, 96], because a transaction is not allowed to release locks until the second phase of the protocol, blocking other transactions and reducing the level of concurrency [42]. In addition, 2PC requires *two network round-trip delays* and *two sequential durable writes* for every distributed transaction, making it a major bottleneck in many distributed transaction processing systems [42]. Although there have been some efforts to eliminate distributed transactions or 2PC, unfortunately, existing solutions either introduce impractical assumptions (e.g., the read/write set of each transaction has to be known a priori in deterministic databases [36, 96, 97]) or significant runtime overhead (e.g., dynamic data partitioning [22, 55]). We will discuss more details on the necessity and cost of 2PC in Section 2.2, and how to reduce the overhead with our proposed solutions in later chapters.

1.1.2 Replication and Consistency

Conventional high availability protocols must make a tradeoff between performance and consistency. On one hand, *asynchronous replication* allows a transaction to commit once its writes arrive at the primary replicas; propagation to backup replicas happens in the background asynchronously [25]. With asynchronous replication, transactions can achieve high performance but a failure may cause data loss, i.e.,

consistency is sacrificed. On the other hand, *synchronous replication* allows a transaction to commit only after its writes arrive at all replicas [19]. No data loss occurs but each transaction holds locks for a longer duration of time and has longer latency — even single-partition transactions need to wait for at least one round-trip of network communication.

1.2 Proposed Solutions

Given two research challenges above, in this thesis, we describe the design of three different systems we have built to address the inefficiency and limitations of existing distributed databases. Each of these systems are designed to reduce the overhead of distributed transactions in OLTP databases, using different mechanisms and bearing various tradeoffs.

- STAR eliminates two-phase commit and network communication used in distributed transactions through asymmetric replication and a phase-switching protocol that dynamically changes the mastership of records. However, it require a single node to hold an entire copy of the database. Note that a single node today can provide tens of terabytes of RAM, which exceeds the scale of all but the most demanding transactional workloads.
- COCO does not make any assumptions on the capacity of each node. It groups transactions arriving within a short time window into epochs, and enforces ACID properties at the granularity of epochs. However, network communication is still needed for replication.
- Aria further reduces the overhead of both 2PC and replication through deterministic execution. By running transaction deterministically across all replicas, each replica does not need to communicate with one another to achieve consistency. In addition, it does not require any analysis or pre-execution of input transactions, making it more practical than existing state-of-the-art deterministic databases. However, unlike the other two systems, Aria may suffer from undesirable performance when a workload has high contention.

Table 1.1: The key characteristics of STAR, COCO and Aria

	STAR	COCO	Aria
Scale out	Sublinear	Linear	Linear
Deterministic	No	No	Yes
Replication cost	Workload dependent	High	Low
Interactive transactions support	Yes (May hurt performance)	Yes (May hurt performance)	No
Performance under high contention	Good	Good	Poor*
Single-node & distributed deployment	Distributed deployment only	Both	Both

*without Aria's fallback strategy

We now briefly introduce our proposed systems, namely, (1) STAR, (2) COCO, and (3) Aria, and highlight our contributions. The key characteristics of each system are summarized in Table 1.1.

1.2.1 STAR

STAR is a new distributed in-memory database based on asymmetric replication. By employing a single-node non-partitioned architecture for some replicas and a partitioned architecture for other replicas, STAR is able to efficiently run both highly partitionable workloads and workloads that involve cross-partition transactions. The key idea is a new *phase-switching* algorithm where the execution of single-partition and cross-partition transactions is separated. In the partitioned phase, single-partition transactions are run on multiple machines in parallel to exploit more concurrency. In the single-master phase, mastership for the entire database is switched to a single designated master node, which can execute these transactions without the use of expensive coordination protocols like two-phase commit. Because the master node has a full copy of the database, this phase-switching can be done at negligible cost.

Contributions: (1) By exploiting multicore parallelism and fast networks, STAR is able to provide high throughput with serializability guarantees. (2) It employs a *phase-switching* scheme which enables STAR to execute cross-partition transactions without two-phase commit while preserving fault tolerance guarantees. (3)It uses a hybrid replication strategy to reduce the overhead of replication, while providing transactional consistency and high-availability.

Assumptions: Our system does require a single node to hold an entire copy of the database. However, cloud service providers now provide high memory instances with 12 TB RAM. Such high memory instances are sufficient to store 10 kilobytes of online state about each customer in a database with about one billion customers, which exceeds the scale of all but the most demanding transactional workloads.

1.2.2 COCO

COCO is a new distributed OLTP database that supports epoch-based commit and replication. The key idea behind COCO is that it separates transactions into epochs and treats a whole epoch of transactions as the commit unit. In this way, the overhead of 2PC and synchronous replication is significantly reduced. We support two variants of optimistic concurrency control (OCC) using physical time and logical time with various optimizations, which are enabled by the epoch-based execution.

Contributions: (1) COCO is the first distributed and replicated main-memory OLTP framework that implements epoch-based commit and replication. (2) We introduce the design of two variants of optimistic concurrency control algorithms in COCO, discuss their tradeoffs in the distributed environment, implement critical performance optimizations, and extend them to support snapshot transactions.

1.2.3 Aria

Aria is a new distributed and deterministic OLTP database that first executes a batch of transactions against the same database snapshot in an *execution phase*, and then deterministically (without communication between replicas) chooses those that should commit to ensure serializability in a *commit phase*. We also propose a novel deterministic reordering mechanism that allows Aria to order transactions in a way that reduces the number of conflicts. In addition, Aria does not require any analysis or pre-execution of input transactions as in existing state-of-the-art deterministic databases [36, 97].

Contributions: (1) Aria supports deterministic transaction execution without any prior knowledge of the input transactions. (2) We introduce a deterministic reordering scheme to reduce conflicts during the commit phase in Aria. (3) We describe a fallback strategy that guarantees Aria is at least as effective as existing deterministic databases in the worst case. Otherwise, a workload may suffer from a high abort rate due to WAW-dependencies, e.g., two transactions update the same record.

1.3 Summary

In this chapter, we introduced the research challenges in distributed and highly available databases, and briefly described our proposed solutions. These three systems are all based on the same design principle — managing transaction processing at the granularity of epochs or batches. STAR uses asynchronous replication within each epoch to enable more efficient replication without sacrificing consistency guarantees. COCO amortizes the cost of two-phase commit and replication by committing or aborting a batch of transactions all together. Aria first runs a batch of transactions and next analyzes the conflicts between transactions within the batch deterministically. The design principle of using epochs or batches enables higher throughput in our proposed solutions and we will discuss in more detail in later chapters.

The rest of this thesis is organized as follows. Chapter 2 provides some background on transaction processing and high availability. We next discuss the three proposed solutions in detail in Chapter 3, Chapter 4 and Chapter 5. The discussion of each of chapter includes a brief introduction, system overview and architecture, description of the proposed system, implementation, experimental evaluation and a discussion of related work. In Chapter 6, we summarize the strengths and limitations of the proposed solutions. Chapter 7 discusses some directions of future work and we conclude the thesis in Chapter 8.

Chapter 2

Background

In this chapter, we present background on concurrency control, replication, and deterministic databases. We begin with a discussion of some popular concurrency control protocols, and the challenges when they are used in distributed databases. We next discuss how database systems achieve high availability through replication. Finally, we discuss how deterministic concurrency control brings a new opportunity for addressing the challenges in both concurrency control and replication.

2.1 Concurrency Control Protocols

Serializability — where the operations of multiple transactions are interleaved to provide concurrency while ensuring that the state of the database is equivalent to some serial ordering of the transactions — is the gold standard for transaction execution. Many serializable concurrency control protocols have been proposed, starting from early protocols that played an important role in hiding disk latency [8] to modern OLTP systems that exploit multicore parallelism [98, 108], typically employing lock-based and/or optimistic techniques.

Two-Phase locking (2PL) is the most widely used classical protocol to ensure serializability of concurrent transactions [34]. 2PL is considered pessimistic since the database acquires locks on operations even when there is no conflict. By contrast, optimistic concurrency control protocols (OCC) avoid this by only checking conflicts

at the end of a transaction's execution [49]. OCC runs transactions in three phases: read, validation, and write. In the read phase, transactions perform read operations from the database and write operations to local copies of objects without acquiring locks. In the validation phase, conflict checks are done against all concurrently committing transactions. If conflicts exist, the transaction aborts. Otherwise, it enters the write phase and copies its local modifications to the database. Modern systems, like Silo [98], typically employ OCC-like techniques because they make it easier to avoid the overhead of shared locks during query execution.

2.2 Concurrency Control in Distributed Databases

We next discuss the challenges when employing concurrency control to distributed databases. In distributed databases, a transaction that accesses data on multiple *participant nodes* is usually initiated and coordinated by a *coordinator*. The most common way to commit transactions in a distributed database is via two-phase commit (2PC) [65]. Once a transaction finishes execution, the coordinator begins the two-phase commit protocol, which consists of a *prepare phase* and a *commit phase*. In the prepare phase, the coordinator communicates with each participant node, asking if it is ready to commit the transaction. To tolerate failures, each participant node must make the prepare/abort decision durable before replying to the coordinator. After the coordinator has collected all votes from participant nodes, it enters the commit phase. In the commit phase, the coordinator is able to commit the transaction if all participant nodes agree to commit. It first makes the commit/abort decision durable and then asks each participant node to commit the transaction. In the end, each participant node acknowledges the commit to the coordinator.

Although the above mechanism ensures both atomicity and durability of distributed transactions, it also introduces some problems that significantly limit the performance of distributed database systems. We now summarize the problems it introduces and discuss the implications for distributed databases: (1) Two network round trips: On top of network round trips during transaction execution, two-phase

commit requires two additional network round trips, making the cost of running a distributed transaction more expensive than the cost of running a single-partition transaction on a single node [74]; (2) Multiple durable writes: A write is considered durable when it has been flushed to disk (e.g., using `fsync` [93]). Depending on different hardware, the latency of a flush is from tens or hundreds of microseconds on SSDs to tens of milliseconds on spinning disks; (3) Increased contention: Multiple network round trips and durable writes also increase the duration that locks are held. As a result, contention increases, which further impairs the throughput and latency.

Some solutions have been proposed to address the inefficiency caused by distributed transactions and 2PC [22, 55, 97], but they all suffer from significant limitations. Schism [22] reduces the number of distributed transactions through a workload-driven partitioning and replication scheme. However, distributed transactions are frequent in real world scenarios and fundamentally unable to be fully partitioned. G-Store [23] and LEAP [55] eliminate distributed transactions by dynamically re-partitioning the database. However, multiple network round trips are still required to move data across a cluster of nodes. Calvin [97] and other deterministic databases avoid 2PC by running transactions deterministically across multiple nodes, but these systems need to know a transaction's read/write set, which is not always feasible.

2.3 Replication in OLTP Databases

A desirable property of any distributed database is high availability, i.e., when a server fails, the system can mask the failure from end users by replacing the failed server with a standby machine. Highly available systems normally replicate data across multiple machines, such that the system still provides some degree of availability when one or more servers fail. We broadly classify existing replication strategies into two categories: (1) asynchronous replication [17, 25], and (2) synchronous replication [46]. In asynchronous replication, data is asynchronously propagated between replicas, meaning a transaction can commit before writes arrive at all replicas. Asynchronous replication reduces the latency to commit transactions, but may suffer from data loss

when failures occur. Instead, in synchronous replication, writes are propagated across all replicas before a transaction commits. Synchronous replication increases latency but ensures strong consistency between replicas and has been a recent focus of the research community [19, 60, 111].

Most transactional systems provide serializability, which requires transactions to produce the results following *some* serial order. Most concurrency control algorithms (e.g., strict two-phase locking [34] and optimistic concurrency control [49]) are non-deterministic, meaning different replicas may diverge when given the same input, because of timing and performance differences on replicas executing transactions in parallel. As a result, some care is needed to guarantee consistency between replicas; most replication protocols adopt either a primary-backup scheme [13] or a state machine replication scheme [50]. In a primary-backup system, the primary runs transactions and ships the results to one or more backups. The backups apply the changes in order to the database and periodically acknowledge the writes back to the primary. A transaction can commit after the writes have been acknowledged by backups. In systems using state machine replication, transactions can run on any replica, but read/write operations need to be sequenced using consensus protocols (e.g., Paxos [50] or Raft [73]). Thus, nondeterministic systems are able to stay consistent but suffer from several limitations, including needing multiple rounds of synchronous communication to commit transactions, and a need to synchronize typically large output values rather than smaller input operations.

2.4 Deterministic Concurrency Control

Deterministic concurrency control algorithms [96] were proposed to address these challenges. In a deterministic database, each replica runs the same set of ordered transactions deterministically, and converts the database from the same initial state to the same final state. Typically, in these systems, transactions first go through a sequencing layer before execution. This sequencing layer acts as the middle layer between clients and the database, and decides a total ordering of transactions submitted

by clients. Usually the sequencing layer runs over multiple machines to avoid having a single point of failure and achieves consensus through a replicated log built on top of Paxos [50] or Raft [73] instances. To avoid non-determinism (e.g., due to system calls to get the current time or generate a random number), the sequencing layer preprocesses incoming transactions to eliminate this non-determinism by replacing such function calls with constant values before passing to replicas. As a result, replicas do not need to communicate with one another to remain consistent, making replication simpler and more efficient. In addition, deterministic databases also reduce the overhead of distributed transactions by avoiding two-phase commit (2PC) [65]. In nondeterministic databases, 2PC is used to ensure that the participants in a distributed transaction reach a single commit/abort decision. In contrast, the failure of one participating node does not affect the commit/abort decision in deterministic databases [96].

Note that existing deterministic databases [35, 36, 96] perform dependency analysis before transaction execution, which requires that the read/write set of a transaction be known a priori.

Chapter 3

STAR: Scaling Transactions through Asymmetric Replication

3.1 Introduction

Recent years have seen a number of in-memory transaction processing systems that can run hundreds of thousands to millions of transactions per second by leveraging multi-core parallelism [91, 98, 108]. These systems can be broadly classified into i) partitioning-based systems, e.g., H-Store [91] which partitions data onto different cores or machines, and ii) non-partitioned systems that try to minimize the overheads associated with concurrency control in a single-server, multi-core setting by avoiding locks and contention whenever possible [70, 72, 98, 108], while allowing any transaction to run on any core/processor.

As shown in Figure 3-1, partitioning-based systems work well when workloads have few cross-partition transactions, because they are able to avoid the use of any synchronization and thus scale out to multiple machines. However, these systems suffer when transactions need to access more than one partition, especially in a distributed setting where expensive protocols like two-phase commit are required to coordinate these cross-partition transactions. In contrast, non-partitioned approaches provide somewhat lower performance on highly-partitionable workloads due to their inability to scale out, but are not sensitive to how partitionable the workload is.

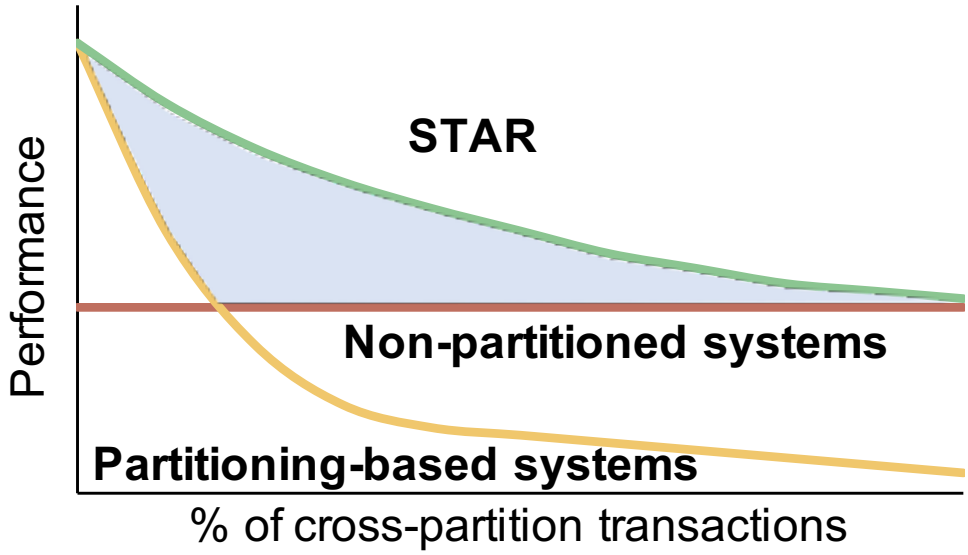


Figure 3-1: Partitioning-based systems vs. Non-partitioned systems

In this chapter, we propose a new transaction processing system, STAR, that is able to achieve the best of both worlds. We start with the observation that most modern transactional systems will keep several replicas of data for high availability purposes. In STAR, we ensure that at least one of these replicas is *complete*, i.e., it stores all records on a single machine, as in the recent non-partitioned approaches. We also ensure that at least one of the replicas is partitioned across several processors or machines, as in partitioned schemes. The system runs in two phases, using a novel *phase-switching* protocol: in the *partitioned* phase, transactions that can be executed completely on a single partition are mastered at one of the partial replicas storing that partition, and replicated to all other replicas to ensure fault tolerance and consistency. Cross-partition transactions are executed during the *single-master* phase, during which mastership for all records is switched to one of the complete replicas, which runs the transactions to completion and replicates their results to the other replicas. Because this node already has a copy of every record, changing the master for a record (*re-mastering*) can be done without transferring any data, ensuring lightweight phase switching. Furthermore, because the node has a complete replica, transactions can be coordinated within a single machine without a slow commit pro-

tocol like two-phase commit. In this way, cross-partition transactions do not incur commit overheads as in typical partitioned approaches (because they all run on the single master), and can be executed using a lightweight concurrency control protocol like Silo [98]. Meanwhile, single-partition transactions can still be executed without any concurrency control at all, as in H-Store [91], and can be run on several machines in parallel to exploit more concurrency. The latency that phase switching incurs is no larger than the typical delay used in high-performance transaction processing systems for an epoch-based group commit.

Although the primary contribution of STAR is this phase-switching protocol, to build the system, we had to explore several novel aspects of in-memory concurrency control. In particular, prior systems like Silo [98] and TicToc [108] were not designed with high-availability (replication) in mind. Making replication work efficiently in these systems requires some care and is amenable to a number of optimizations. For example, STAR uses intra-phase asynchronous replication to achieve high performance. In the meantime, it ensures consistency among replicas via a replication fence when phase-switching occurs. In addition, with our phase-switching protocol, STAR can use a cheaper replication strategy than that employed by replicated systems that need to replicate entire records [113]. This optimization can significantly reduce bandwidth requirements (e.g., by up to an order of magnitude in our experiments with TPC-C).

Our system does require a single node to hold an entire copy of the database, as in many other modern transactional systems [29, 36, 47, 54, 98, 102, 108]. Cloud service providers, such as Amazon EC2 [2] and Google Cloud [3], now provide high memory instances with 12 TB RAM. Such high memory instances are sufficient to store 10 kilobytes of online state about each customer in a database with about one billion customers, which exceeds the scale of all but the most demanding transactional workloads. In addition, on workloads with skew, a single-node in-memory database can be further extended through an anti-caching architecture [24], which provides 10x larger storage and competitive performance to in-memory databases. The single node in STAR may become a major bottleneck in workloads that have high replication

cost, however, the same bottleneck exists in existing non-partitioned systems as well, making STAR perform at least as effective as non-partitioned systems in the worst case.

3.2 STAR Architecture

STAR is a distributed and replicated in-memory database. It separates the execution of single-partition transactions and cross-partition transactions using a novel *phase-switching protocol*. The system dynamically switches between *partitioned* and *single-master* phases. In STAR, each partition is mastered by a single node. During the partitioned phase, queries that touch only a single partition are executed one at a time to completion on their partition. Queries that touch multiple partitions (even if those partitions are on a single machine) are executed in the single-master phase on a single designated master node.

To support the phase-switching protocol, STAR employs asymmetric replication. As shown in Figure 3-2, the system consists of two types of replicas, namely, (1) full replicas, and (2) partial replicas. Each of the f nodes (left side of figure) has a full replica, which consists of all database partitions. Each of the k nodes (right side of figure) has a partial replica, which consists of a portion of the database. In addition, STAR requires that these k partial replicas together contain at least one full copy of the database. During the partitioned phase, each node (whether a full or partial replica) acts as a master for some part of the database. During the single-master phase, one of the f nodes acts as the master for the whole database. Note that writes of committed transactions are replicated at least $f + 1$ times on a cluster of $f + k$ nodes. We envision f being small (e.g., 1), while k can be much larger. Having more than one full replica (i.e., $f > 1$) does not necessarily improve system performance, but provides higher availability when failures occur (See Section 3.3.5).

There are several advantages of this phase switching approach. First, in the single-master phase, cross-partition transactions are run on a single master node. As in existing non-partitioned systems, two-phase commit is not needed as the master

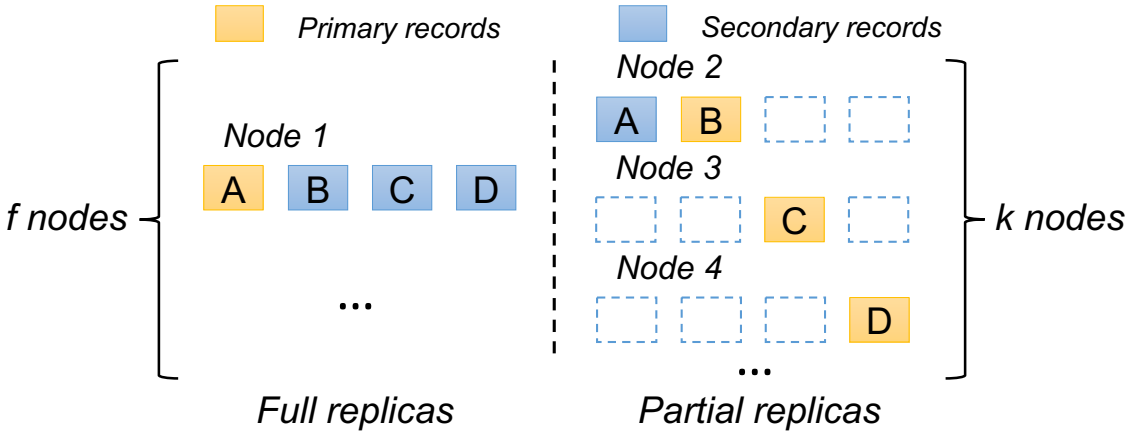


Figure 3-2: The architecture of STAR

node runs all cross-partition transactions and propagates writes to replicas. In contrast, transactions involving multiple nodes in existing distributed partitioning-based systems are coordinated with two-phase commit, which can significantly limit the throughput of distributed database systems [42]. Note that transactions running in the partitioned phase also do not require two-phase commit because they all run on a single partition, on a single node. Second, in the partitioned phase, single-partition transactions are run on multiple nodes in parallel, providing more concurrency, as in existing partitioning-based systems. This asymmetric replication approach is a good fit for workloads with a mix of single-partition and cross-partition transactions on a cluster of about 10 nodes. We demonstrate this both empirically and through the use of an analytical model, as shown in Sections 3.6 and 3.5.3. Figure 3-3 shows the speedup predicted by our model of STAR with n nodes over a single node.

STAR uses a variant of Silo's OCC protocol [98]. Each record in the database maintains a transaction ID (TID) from the transaction that last updated the record. The TID is used to determine which other transactions a committing transaction may have a conflict with. For example, once a transaction begins validation, it will abort if any one of the accessed records is locked by other transactions or has a different TID. The TID is assigned to a transaction after it is successfully validated. There are three criteria for the TID obtained from each thread: (a) it must be larger than the TID of any record in the read/write set; (b) it must be larger than the thread's last chosen

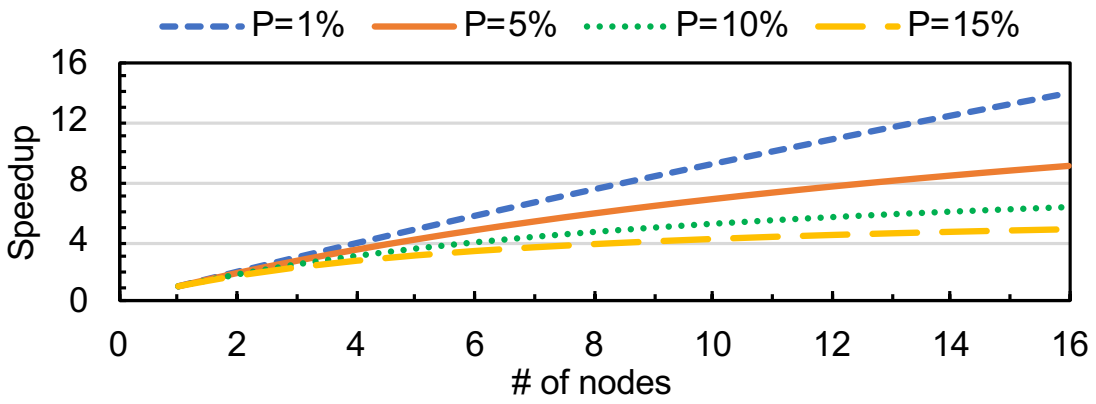


Figure 3-3: Performance speedup of asymmetric replication in STAR over single node execution; $\mathcal{P}$ = Percentage of cross-partition transactions

TID; (c) it must be in the current global epoch. The first two criteria guarantee that the TIDs of transactions having conflicting writes are assigned in a serial-equivalent order. As in Silo, STAR uses a form of *epoch-based group commit*. The serial order of transactions running within an epoch is not explicitly known by the system except across epoch boundaries, since anti-dependencies (i.e., write-after-read conflicts) are not tracked. Different from Silo, in which the global epoch is incremented every 40 ms, a phase switch in STAR naturally forms an epoch boundary. Therefore, the system only releases the result of a transaction when the next phase switch occurs. At this time, the global epoch is also incremented.

At any point in time, each record is *mastered* on one node, with other nodes serving as secondaries. Transactions are only executed over primary records; writes of committed transactions are propagated to all replicas. One way to achieve consistency is to replicate writes *synchronously* among replicas. In other words, the primary node holds the write locks when replicating writes of a committed transaction to backup nodes. However, this design increases the latency of replication and impairs system performance. In STAR, writes of committed transactions are buffered and *asynchronously* shipped to replicas, meaning the locks on the primary are not held during the process. To ensure correctness, we employ the Thomas write rule [95]: we tag each record with the last TID that wrote it and apply a write if the TID of the write is larger than the current TID of the record. Because TIDs of conflicting writes

are guaranteed to be assigned in the serial-equivalent order of the writing transactions, this rule will guarantee correctness. In addition, STAR uses a *replication fence* when phase-switching occurs. In this way, strong consistency among replicas is ensured across the boundaries of the partitioned phase and the single-master phase.

Tables in STAR are implemented as collections of hash tables, which is typical in many in-memory databases [102, 105, 108]. Each table is built on top of a primary hash table and contains zero or more secondary hash tables as secondary indexes. To access a record, STAR probes the hash table with the primary key. Fields with secondary indexes can be accessed by mapping a value to the relevant primary key. Although STAR is built on top of hash tables, it is easily adaptable to other data structures such as Masstree [62]. As in most high performance transactional systems [91, 98, 108], clients send requests to STAR by calling pre-defined *stored procedures*. Parameters of a stored procedure must also be passed to STAR with the request. Arbitrary logic (e.g., read/write operations) is supported in stored procedures, which are implemented in C++.

STAR is *serializable* by default but can also support *read committed* and *snapshot isolation*. A transaction runs under *read committed* by skipping read validation on commit, since STAR uses OCC and uncommitted data never occurs in the database. STAR can provide snapshot isolation by retaining additional versions for records [98].

3.3 The Phase Switching Algorithm

We now describe the phase switching algorithm we use to separate the execution of single-partition and cross-partition transactions. We first describe the two phases and how the system transitions between them. We next give a brief proof to show that STAR produces serializable results. In the end, we discuss how STAR achieves fault tolerance and recovers when failures occur.

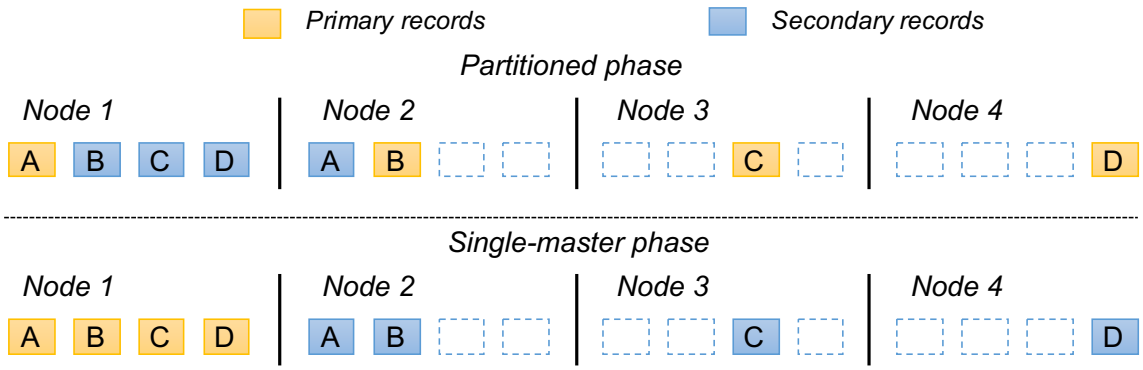


Figure 3-4: Illustrating the two execution phases

3.3.1 Partitioned Phase Execution

Each node serves as primary for a subset of the records in the partitioned phase, as shown on the top of Figure 3-4. During this phase, we restrict the system to run transactions that only read from and write into a single partition. Cross-partition transactions are deferred for later execution in the single-master phase.

In the partitioned phase, each partition is touched only by a single worker thread. A transaction keeps a read set and a write set in its local copy, in case a transaction is aborted by the application explicitly. For example, an invalid item ID may be generated in TPC-C and a transaction with an invalid item ID is supposed to abort during execution. At commit time, it's not necessary to lock any record in the write set and do read validation, since there are no concurrent accesses to a partition in the partitioned phase. The system still generates a TID for each transaction and uses the TID to tag the updated records. In addition, writes of committed transactions are replicated to other backup nodes.

3.3.2 Single-Master Phase Execution

Any transaction can run in the single-master phase. Threads on the designated master node can access any record in any partition, since it has become the primary for all records. For example, node 1 is the master node on the bottom of Figure 3-4. We use multiple threads to run transactions using a variant of Silo's OCC protocol [98] in the single-master phase.

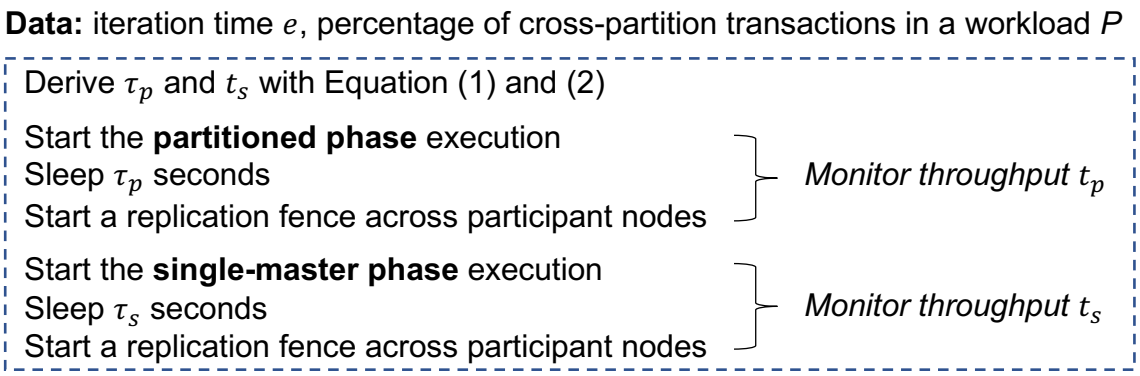


Figure 3-5: Phase transitions in STAR

A transaction reads data and the associated TIDs and keeps them in its read set for later read validation. During transaction execution, a write set is computed and kept in a local copy. At commit time, each record in the write set is locked in a global order (e.g, the addresses of records) to prevent deadlocks. The transaction next generates a TID based on its read set, write set and the current epoch number. The transaction will abort during read validation if any record in the read set is modified (by comparing TIDs in the read set) or locked. Finally, records are updated, tagged with the new TID, and unlocked. After the transaction commits, the system replicates its write set to other backup nodes.

Note that fault tolerance must satisfy a transitive property [98]. The result of a transaction can only be released to clients after its writes and the writes of all transactions serialized before it are replicated. In STAR, the system treats each epoch as a commit unit through a replication fence. By doing this, it is guaranteed that the serial order of transactions is always consistent with the epoch boundaries. Therefore, the system does not release the results of transactions to clients until the next phase switch (and epoch boundary) occurs.

3.3.3 Phase Transitions

We now describe how STAR transitions between the two phases, which alternate after the system starts. The system starts in the partitioned phase, deferring cross-partition transactions for later execution.

For ease of presentation, we assume that all cross-partition transaction requests go to the designated master node (selected from among the f nodes with a full copy of the database). Similarly, each single-partition transaction request only goes to the participant node that has the mastership of the partition. In the single-master phase, the master node becomes the primary for all records. In the partitioned phase, the master node acts like other participant nodes to run single-partition transactions. This could be implemented via router nodes that are aware of the partitioning of the database. If some transaction accesses multiple partitions on a non-master node, the system would re-route the request to the master node for later execution.

In the partitioned phase, a thread on each partition fetches requests from clients and runs these transactions as discussed in Section 3.3.1. When the execution time in the partitioned phase exceeds a given threshold τ_p, STAR switches all nodes into the single-master phase, as shown in Figure 3-5. The phase switching algorithm is coordinated by a stand-alone *coordinator* outside of STAR instances. It can be deployed on any node of STAR or on a different node for better availability.

Before the phase switching occurs, the coordinator in STAR stops all worker threads. During a replication fence, all participant nodes synchronize statistics about the number of committed transactions with one another. From these statistics each node learns how many outstanding writes it is waiting to see; nodes then wait until they have received and applied all writes from the replication stream to their local database. Finally, the coordinator switches the system to the other phase.

In the single-master phase, worker threads on the master node pull requests from clients and run transactions as discussed in Section 3.3.2. Meanwhile, the master node sends writes of committed transactions to replicas and all the other participant nodes stand by for replication. To further improve the utilization of servers, read-only transactions can run under read committed isolation level on non-master nodes at the client's discretion. Once the execution time in the single-master phase exceeds a given threshold τ_s, the system switches back to the partitioned phase using another replication fence.

The parameters τ_p and τ_s are set dynamically according to the system's throughput

t_p in the partitioned phase, the system's throughput t_s in the single-master phase, the percentage $\mathcal{P}$ of cross-partition transactions in the workload, and the iteration time e.

$$\tau_p + \tau_s = e \tag{3.1}$$

$$\frac{\tau_s t_s}{\tau_p t_p + \tau_s t_s} = \mathcal{P} \tag{3.2}$$

Note that t_p, t_s and $\mathcal{P}$ are monitored and collected by the system in real time, and e is supplied by the user. Thus, these equations can be used to solve for τ_p and τ_s, as these are the only two unknowns. When there are no cross-partition transactions (i.e., $\mathcal{P} = 0$ and t_s is not well-defined), the system sets τ_p to e and τ_s to 0.

Intuitively, the system spends less time on synchronization with a longer iteration time e. In our experiments, we set the default iteration time to 10 ms; this provides good throughput while keeping latency at a typical level for high throughput transaction processing systems (e.g., Silo [98] uses 40 ms as a default).

Note that this deferral-based approach is symmetric so that single-partition transactions have the same expected mean latency as cross-partition transactions regardless of the iteration time (i.e., $\tau_p + \tau_s$), assuming all transactions arrive at a uniform rate. For a transaction, the latency depends on when the phase in which it is going to run ends. The mean latency is expected to be $(\tau_p + \tau_s)/2$.

3.3.4 Serializability

We now give a brief argument that transactions executed in STAR are serializable. A transaction only executes in a single phase, i.e., it runs in either the partitioned phase or the single-master phase. A replication fence between the partitioned phase and the single-master phase ensures that all writes from the replication stream have been applied to the database before switching to the next phase.

In the partitioned phase, there is only one thread running transactions serially on each partition. Each executed transaction only touches one partition, which makes transactions clearly serializable. In the single-master phase, STAR implements a vari-

ant of Silo's OCC protocol [98] to ensure that concurrent transactions are serializable. With the Thomas write rule, the secondary records are correctly synchronized, even though the log entries from the replication stream may be applied in an arbitrary order.

3.3.5 Fault Tolerance

In-memory replication provides high availability to STAR, since transactions can run on other active nodes even though some nodes failed. To prevent data loss and achieve durability, the system must log writes of committed transactions to disk. Otherwise, data will be lost when all replicas fail (e.g., due to power outage).

In this section, we first describe how STAR achieves durability via disk logging and checkpointing and then introduce how failures are detected. Finally, we discuss how STAR recovers from failures under different scenarios.

Disk logging

In STAR, each worker thread has a local recovery log. The writes of committed transactions along with some metadata are buffered in memory and periodically flushed to the log. Specifically, a log entry contains a single write to a record in the database, which has the following information: (1) key, (2) value, and (3) TID. The TID is from the transaction that last updated the record, and has an embedded epoch number as well. The worker thread periodically flushes buffered logs to disk; STAR also flushes all buffers to disk in the replication fence.

To bound the recovery time, a dedicated checkpointing thread can be used to periodically checkpoint the database to disk as well. The checkpointing thread scans the database and logs each record along with the TID to disk. A checkpoint also records the epoch number e_c when it starts. Once a checkpoint finishes, all logs earlier than epoch e_c can be safely deleted. Note that a checkpoint does not need to be a consistent snapshot of the database, as in SiloR [113], allowing the system to not freeze during checkpointing. On recovery, STAR uses the logs since the checkpoint

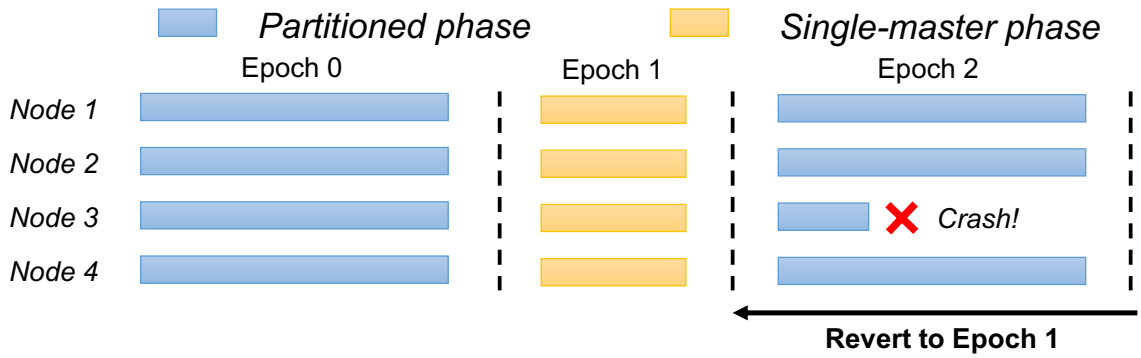


Figure 3-6: Failure detection in replication fence

(i.e., e_c) to correct the inconsistent snapshot with the Thomas write rule.

Failure detection

Before introducing how STAR detects failures, we first give some definitions and assumptions on failures. In this thesis, we assume fail-stop failures. A *healthy node* is a node that can connect to the coordinator, accept a client's requests, run transactions and replicate writes to other nodes. A *failed node* is one on which the process of a STAR instance has crashed [25] or which cannot communicate over the network.

The coordinator detects failures during the replication fence. If all nodes successfully finish the replication fence, the coordinator writes a log record to disk, which indicates the commit point of the current epoch. When an epoch commits, the system makes the writes of committed transactions visible to clients and guarantees that all writes of committed transactions have been logged to disk and the data across replicas is consistent. Note that the system always recovers to the last committed epoch if a failure occurs. If some node does not respond to the coordinator, it is considered to be a failed node. The list of failed nodes is broadcast to all healthy participant nodes in STAR. In this way, healthy nodes can safely ignore all replication messages from failed nodes that have lost network connectivity to the coordinator. Thus, the coordinator acts as a view service to solve the "split brain" problem, coordinating data movement across nodes on failures. To prevent the coordinator from being a single point of failure, it can be implemented as a replicated state machine with Paxos [50]

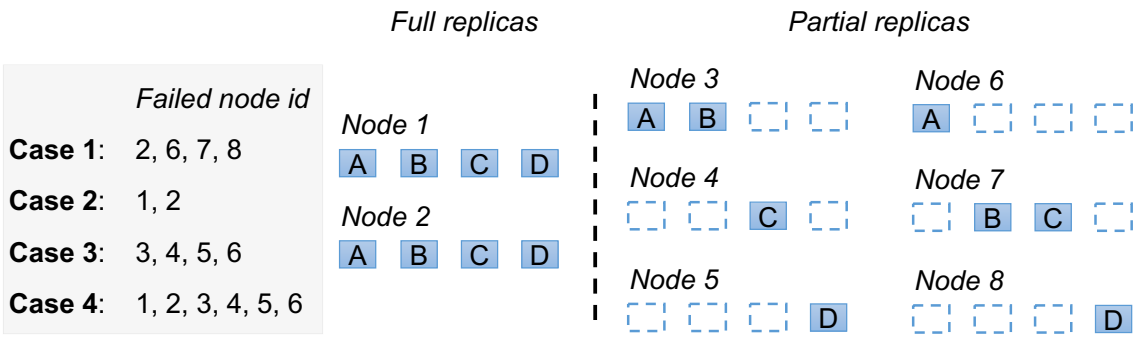


Figure 3-7: Illustrating different failure scenarios

or Raft [73].

Once a failure is detected by the coordinator, the system enters recovery mode and reverts the database to the last committed epoch, as shown in Figure 3-6. To achieve this, the database maintains two versions of each record. One is the most recent version prior to the current phase and the other one is the latest version written in the current phase. The system ignores all data versions written in the current phase, since they have not been committed by the database.

We next describe how STAR recovers from failures once the database has been reverted to a consistent snapshot.

Recovery

We use examples from Figure 3-7 to discuss how STAR recovers from failures. In these examples, there are 2 nodes with full replicas and 6 nodes with partial replicas (i.e., $f = 2$ and $k = 6$). A cluster of 8 nodes could fail in $2^8 - 1 = 255$ different ways, which fall into the following four different scenarios. Here, a "full replica" is a replica on a single node, and a "complete partial replica" is a set of partial replicas that collectively store a copy of the entire database.

(1) At least one full replica and one complete partial replica remain.

(2) No full replicas remain but at least one complete partial replica remains.

(3) No complete partial replicas remain but at least one full replica remains.

(4) No full replicas or complete partial replicas remain.

We now describe how STAR recovers under each scenario.

Case 1: As shown in Figure 3-7, failures occur on nodes 2, 6, 7 and 8. The system can still run transactions with the phase-switching algorithm. When a failed node recovers, it copies data from remote nodes and applies them to its database. In parallel, it processes updates from the relevant currently healthy nodes using the Thomas write rule. Once all failed nodes finish recovery, the system goes back to the normal execution mode.

Case 2: If no full replicas are available, the system falls back to a mode in which a distributed concurrency control algorithm is employed, as in distributed partitioning-based systems (e.g., Dist. OCC). The recovery process on failed nodes is the same as in Case 1.

Case 3: If no complete partial replicas are available, the system can still run transactions with the phase-switching algorithm. However, the mastership of records on lost partitions have to be reassigned to the nodes with full replicas. If all nodes with partial replicas fail, the system runs transactions only on full replicas without the phase-switching algorithm. The recovery process on failed nodes is the same as in Case 1.

Case 4: The system stops processing transactions (i.e., loss of availability) when no complete replicas remain. Each crashed node loads the most recent checkpoint from disk and restores its database state to the end of the last epoch by replaying the logs since the checkpoint with the Thomas write rule. The system goes back to the normal execution mode once all nodes finish recovery.

Note that STAR also supports recovery from nested failures. For example, an additional failure on node 3 could occur during the recovery of Case 1. The system simply reverts to the last committed epoch and begins recovery as described in Case 3.

3.4 Replication: Value vs. Operation

In this section, we describe the details of our replication schemes, and how replication is done depending on the execution phase. As discussed earlier, STAR runs single-partition and cross-partition transactions in different phases. The system uses different replication schemes in these two phases: in the single-master phase, because a partition can be updated by multiple threads, records need to be fully-replicated to all replicas to ensure correct replication. However, in the partitioned phase, where a partition is only updated by a single thread, the system can use a better replication strategy based on replicating operations to improve performance. STAR provides APIs for users to manually program the operations, e.g., string concatenation.

To illustrate this, consider two transactions being run by two threads: `T1: R1.A = R1.B + 1; R2.C = 0` and `T2: R1.B = R1.A + 1; R2.C = 1`. Suppose `R1` and `R2` are two records from different partitions and we are running in the single-master phase. In this case, because the writes are done by different threads, the order in which the writes arrive on replicas may be different from the order in which transactions commit on the primary. To ensure correctness, we employ the Thomas write rule: apply a write if the TID of the write is larger than the current TID of the record. However, for this rule to work, each write must include the values of all fields in the record, not just the updated fields. To see this, consider the example in the left side of Figure 3-8 (only R1 is shown); For record `R1`, if `T1` only replicates `A`, `T2` only replicates `B`, and `T2`'s updates are applied before `T1`'s, transaction `T1`'s update to field `A` is lost, since `T1` is less than `T2`. Thus, when a partition can be updated by multiple threads, all fields of a record have to be replicated as shown in the middle of Figure 3-8. Note that fields that are always read-only do not need to be replicated.

Now, suppose `R1` and `R2` are from the same partition, and we run the same transactions in the partitioned phase, where transactions are run by only a single thread on each partition. If `T2` commits after `T1`, `T1` is guaranteed to be ahead of `T2` in the replication stream since they are executed by the same thread. For this reason, only the updated fields need to be replicated, i.e., `T1` can just send the new value for `A`,

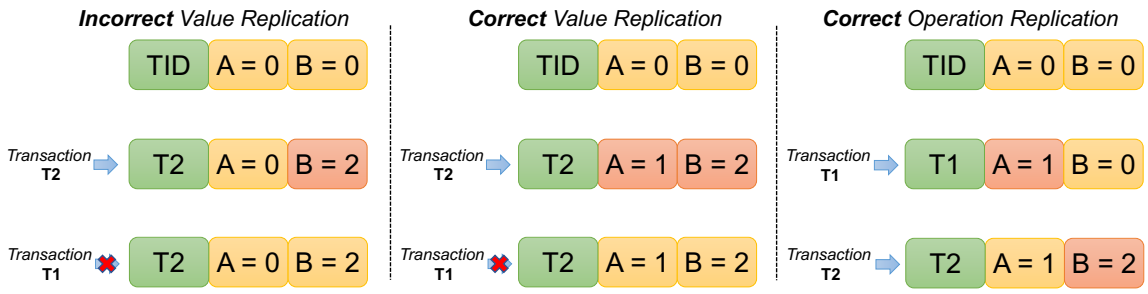


Figure 3-8: Illustrating different replication schemes; Red rectangle shows an updated field

and `T2` can just send the new value for `B` as shown in the right side of Figure 3-8. Furthermore, in this case, the system can also choose to replicate the *operation* made to a field instead of the value of a field in a record. This can significantly reduce the amount of data that must be sent. For example, in the `Payment` transaction in TPC-C, a string is concatenated to a field with a 500-character string in `Customer` table. With operation replication, the system only needs to replicate a short string and can re-compute the concatenated string on each replica, which is much less expensive than sending a 500-character string over network. This optimization can result in an order-of-magnitude reductions in replication cost.

In STAR, a *hybrid replication* strategy is used, i.e., the master node uses value replication strategy in the single-master phase and all nodes use the operation replication strategy in the partitioned phase. The hybrid strategy achieves the best of of both worlds: (1) value replication enables out-of-order replication, not requiring a serial order which becomes a bottleneck in the single master phase (See Section 3.6.5), and (2) in the partitioned phase, operation replication reduces the communication cost compared to value replication, which always replicates the values of all fields in a record.

As discussed earlier, STAR logs the writes of committed transactions to disk for durability. The writes can come from either local transactions or remote transactions through replication messages. By default, STAR logs the whole record to disk for fast and parallel recovery. However, a replication message in operation replication only has operations rather than the value of a whole record. Consider the example

in the right side of Figure 3-8. The replication messages only have `T1: A = 1` and `T2: B = 2`. To solve this problem, when a worker thread processes a replication message that contains an operation, it first applies the operation to the database and then copies the value of the whole record to its logging buffer. In other words, the replication messages are transformed into `T1: A = 1; B = 0` and `T2: A = 1; B = 2` before logging to disk. By doing this, the logs can still be replayed in any order with the Thomas write rule during recovery.

3.5 Discussion

We now discuss the tradeoffs that non-partitioned systems and partitioning-based systems achieve and use an analytical model to show how STAR achieves the best of both worlds.

3.5.1 Non-partitioned Systems

A typical approach to build a fault tolerant non-partitioned system is to adopt the primary/backup model. A primary node runs transactions and replicates writes of committed transactions to one or more backup nodes. If the primary node fails, one of the backup nodes can take over immediately without loss of availability.

As we show in Table 3.1, the writes of committed transactions can be replicated from the primary node to backup nodes either synchronously or asynchronously. With synchronous replication, a transaction releases its write locks and commits as soon as the writes are replicated (low commit latency), however, round trip communication is needed even for single-partition transactions (high write latency). With asynchronous replication, it's not necessary to hold write locks on the primary node during replication, and the writes may be applied in any order on backup nodes with value replication and the Thomas write rule. To address the potential inconsistency issue when a fault occurs, an epoch-based group commit (high commit latency) must be used as well. The epoch-based group commit serves as a barrier that guarantees all writes are replicated when transactions commit. Asynchronous replication reduces

Table 3.1: Overview of each approach; SYNC: synchronous replication; ASYNC: asynchronous replication + epoch-based group commit

		Non-partitioned		Partitioning-based	
	STAR	SYNC	ASYNC	SYNC	ASYNC
Write latency	Low	High	Low	High	Medium
Commit latency	High	Low	High	Low	High
Scale out	Medium	Low	Low	High	High
Performance sensitivity to cross-partition transactions	Low	Low	Low	High	High
Replication strategy	Hybrid	Operation	Value	Operation	Value

the amount of time that a transaction holds write locks during replication (low write latency) but incurs high commit latency for all transactions.

The performance of non-partitioned systems has low sensitivity to cross-partition transactions in a workload, but they cannot easily scale out. The CPU resources on backup nodes are often under-utilized, using more hardware to provide a lower overall throughput.

3.5.2 Partitioning-based Systems

In partitioning-based systems, the database is partitioned in a way such that each node owns one or more partitions. Each transaction has access to one or more partitions and commits with distributed concurrency control protocols (e.g., strict two-phase locking) and 2PC. This approach is a good fit for workloads that have a natural partitioning as the database can be treated as many disjoint sub-databases. However, cross-partition transactions are frequent in real-world scenarios. For example, in the standard mix of TPC-C, 10% of `NewOrder` and 15% of `Payment` are cross-partition transactions.

The same primary/backup model as in non-partitioned systems can be utilized to make partitioning-based systems fault tolerant. With synchronous replication, the write latency of partitioning-based systems is the same as non-partitioned systems. With asynchronous replication, the write latency depends on the number of partitions

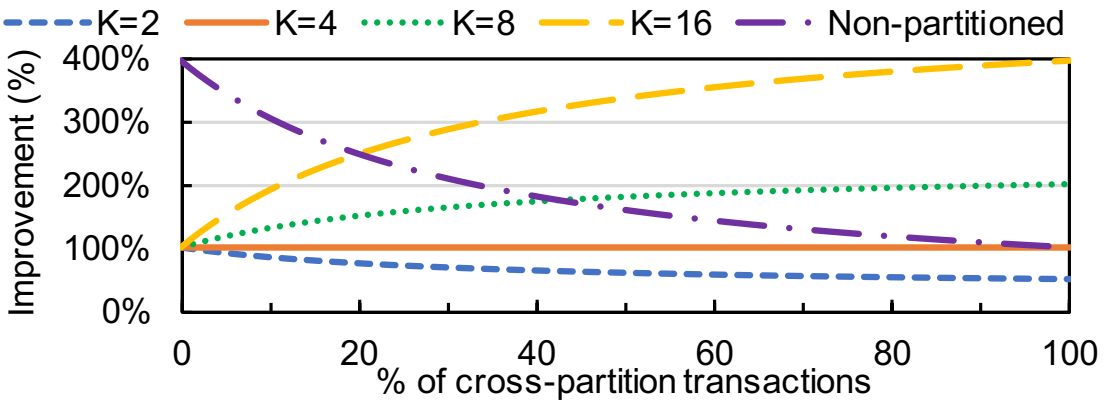


Figure 3-9: Illustrating effectiveness of STAR, vs. partitioning based systems for varying levels of $\mathcal{K}$, and against non-partitioned systems

each transaction updates and on variance of communication delays.

If all transactions are single-partition transactions, partitioning-based systems are able to achieve linear scalability. However, even with a small fraction of cross-partition transactions, partitioning-based systems suffer from high round trip communication cost such as remote reads and distributed commit protocols.

3.5.3 Achieving the Best of Both Worlds

We now use an analytical model to show how STAR achieves the best of both worlds. Suppose we have a workload with n_s single-partition transactions and n_c cross-partition transactions. We first analyze the time to run a workload with a partitioning-based approach on a cluster of n nodes. If the average time of running a single-partition transaction and a cross-partition transaction in a partitioning-based system is t_s and t_c seconds respectively, we have

$$T_{\text{Partitioning-based}}(n) = (n_s t_s + n_c t_c)/n \tag{3.3}$$

In contrast, the average time of running a cross-partition transaction is almost the same as running a single-partition transaction in a non-partitioned approach (e.g., in

a primary-backup database), and therefore,

$$T_{\text{Non-partitioned}}(n) = (n_s + n_c)t_s \tag{3.4}$$

In STAR, single-partition transactions are run on all participant nodes, and cross-partition transactions are run with a single master node. If the replication and phase transitions are not bottlenecks, we have

$$T_{\text{STAR}}(n) = (n_s/n + n_c)t_s \tag{3.5}$$

We let $\mathcal{K} = t_c/t_s$ and $\mathcal{P} = n_c/(n_c+n_s)$. Thus, $\mathcal{K}$ indicates how much more expensive a cross-partition transaction is than a single-partition transaction, and $\mathcal{P}$ indicates the percentage of cross-partition transactions in a workload. We now give the performance improvement that STAR achieves over the other two approaches,

$$\begin{aligned} I_{\text{Partitioning-based}}(n) &= \frac{T_{\text{Partitioning-based}}(n)}{T_{\text{STAR}}(n)} = \frac{\mathcal{K}\mathcal{P} - \mathcal{P} + 1}{n\mathcal{P} - \mathcal{P} + 1} \\ I_{\text{Non-partitioned}}(n) &= \frac{T_{\text{Non-partitioned}}(n)}{T_{\text{STAR}}(n)} = \frac{n}{n\mathcal{P} - \mathcal{P} + 1} \end{aligned}$$

Similarly, we have the scalability of asymmetric replication by showing the speedup that STAR achieves with n nodes over a single node,

$$I(n) = \frac{T_{\text{STAR}}(1)}{T_{\text{STAR}}(n)} = \frac{n}{n\mathcal{P} - \mathcal{P} + 1}$$

For different values of $\mathcal{K}$, we plot $I_{\text{Partitioning-based}}(4)$ and $I_{\text{Non-partitioned}}(4)$ in Figure 3-9, when varying the percentage of cross-partition transactions on a cluster of four nodes. STAR outperforms non-partitioned systems as long as there are single-partition transactions in a workload. This is because all single-partition transactions are run on all participant nodes, which makes the system utilize more CPU resources from multiple nodes. To outperform partitioning-based systems, the average time of running a cross-partition transaction must exceed n times of the average time to run a single-partition transaction (i.e., $\mathcal{K} > n$).

3.6 Evaluation

In this section, we evaluate the performance of STAR focusing on the following key questions:

- How does STAR perform compared to non-partitioned systems and partitioning-based systems?
- How does STAR perform compared to deterministic databases?
- How does the phase switching algorithm affect the throughput of STAR and what's the overhead?
- How effective is STAR's replication strategy?
- How does STAR scale?

3.6.1 Experimental Setup

We ran our experiments on a cluster of four `m5.4xlarge` nodes running on Amazon EC2 [2]. Each node has 16 2.50 GHz virtual CPUs and 64 GB of DRAM running 64-bit Ubuntu 18.04. `iperf` shows that the network between each node delivers about 4.8 Gbits/s throughput. We implemented STAR and other distributed concurrency control algorithms in C++. The system is compiled using GCC 7.3.0 with `-O2` option enabled.

In our experiments, we run 12 worker threads on each node, yielding a total of 48 worker threads. Each node also has 2 threads for network communication. We made the number of partitions equal to the total number of worker threads. All results are the average of three runs. We ran transactions at the serializability isolation level.

Workloads

To study the performance of STAR, we ran a number of experiments using two popular benchmarks:

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) is a simple transactional workload designed to facilitate performance comparisons of database and key-value systems [18]. It has a single table with 10 columns. The primary key of each record is a

64-bit integer and each column consists of 10 random bytes. A transaction accesses 10 records and each access follows a uniform distribution. We set the number of records to 200K per partition, and we run a workload mix of 90/10, i.e., each transaction has 9 read operations and 1 read/write operation. By default, we run this workload with 10% cross-partition transactions that access to multiple partitions.

TPC-C: The TPC-C benchmark [1] is the gold standard for evaluating OLTP databases. It models a warehouse order processing system, which simulates the activities found in complex OLTP applications. It has nine tables and we partition all the tables by `Warehouse ID`. We support two transactions in TPC-C, namely, (1) `NewOrder` and (2) `Payment`. 88% of the standard TPC-C mix consists of these two transactions. The other three transactions require range scans, which are currently not supported in our system. By default, we ran this workload with the standard mix, in which a `NewOrder` transaction is followed by a `Payment` transaction. By default, 10% of NewOrder and 15% of Payment transactions are cross-partition transactions that access multiple warehouses.

In YCSB, each partition adds about 25 MB to the database. In TPC-C, each partition contains one warehouse and adds about 100 MB to the database.

To measure the maximum throughput that each approach can achieve, every worker thread generates and runs a transaction to completion one after another in our experiments.

Concurrency control algorithms

To avoid an apples-to-oranges comparison, we implemented each of the following concurrency control algorithms in C++ in our framework.

STAR: This is our algorithm as discussed in Section 3.2. We set the iteration time of a phase switch to 10 ms. To have a fair comparison to other algorithms, disk logging, checkpointing, and the hybrid replication optimization are disabled unless otherwise stated.

PB. OCC: This is a variant of Silo's OCC protocol [98] adapted for a primary/backup setting. The primary node runs all transactions and replicates the

writes to the backup node. Only two nodes are used in this setting.

Dist. OCC: This is a distributed optimistic concurrency control protocol. A transaction reads from the database and maintains a local write set in the execution phase. The transaction first acquires all write locks and next validates all reads. Finally, it applies the writes to the database and releases the write locks.

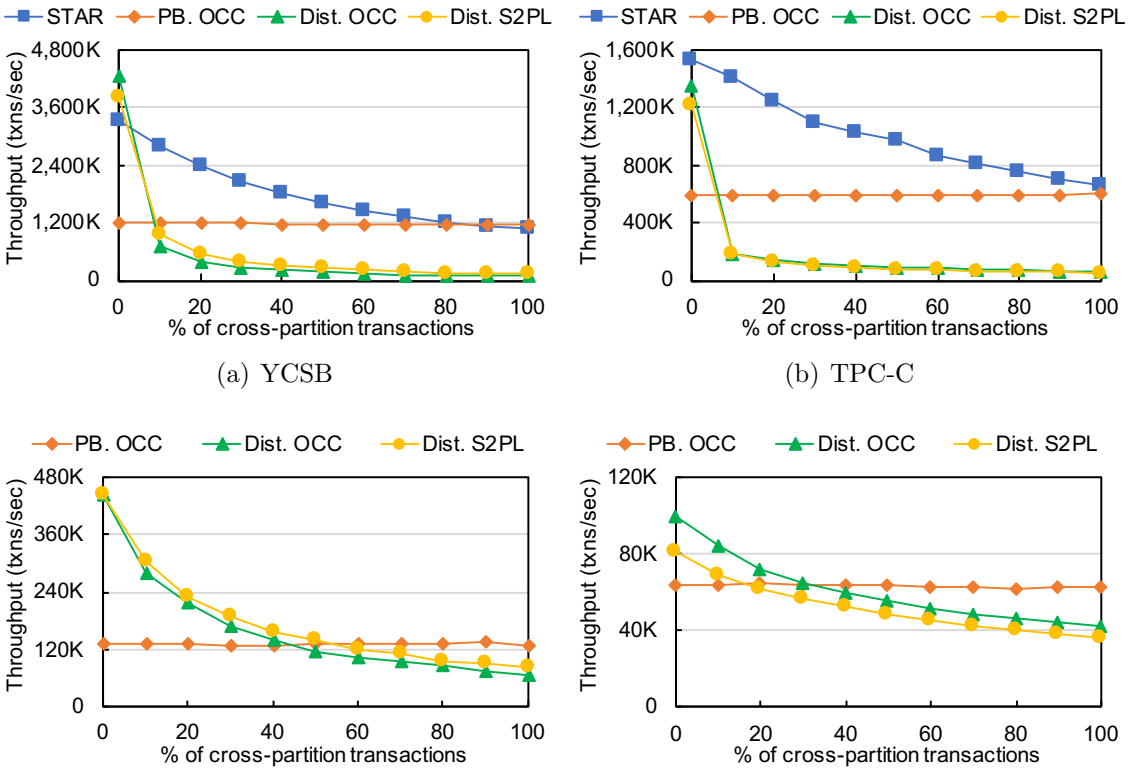
Dist. S2PL: This is a distributed strict two-phase locking protocol. A transaction acquires read and write locks during execution. The transaction next executes to compute the value of each write. Finally, it applies the writes to the database and releases all acquired locks.

In our experiments, PB. OCC is a non-partitioned system, and Dist. OCC and Dist. S2PL are considered as partitioning-based systems. We use `NO_WAIT` policy to avoid deadlocks in partitioning-based systems, i.e., a transaction aborts if it fails to acquire some lock. This deadlock prevention strategy was shown to be the most scalable protocol [42]. We do not report the results on PB. S2PL, since it always performs worse than PB. OCC [108]. Also note that we added an implementation of Calvin, described in Section 3.6.3.

Partitioning and replication configuration

In our experiment, we set the number of replicas of each partition to 2. Each partition is assigned to a node by a hash function. The primary partition and secondary partition are always hashed to two different nodes. In STAR, we have 1 node with full replica and 3 nodes with partial replica, i.e., $f = 1$ and $k = 3$. Each node masters a different portion of the database, as shown in Figure 3-2.

We consider two variations of PB. OCC, Dist. OCC, and Dist. S2PL: (1) asynchronous replication and epoch-based group commit, and (2) synchronous replication. Note that Dist. OCC and Dist. S2PL must use two-phase commit when synchronous replication is used. In addition, synchronous replication requires that all transactions hold write locks during the round trip communication for replication.



(a) YCSB

(b) TPC-C

(c) YCSB w/ Sync. Rep.

(d) TPC-C w/ Sync. Rep.

Figure 3-10: Performance and latency comparison of each approach on YCSB and TPC-C

3.6.2 Performance Comparison

We now compare STAR with a non-partitioned system and two partitioning-based systems using both YCSB and TPC-C workloads.

Results of asynchronous replication and epoch-based group commit

We ran both YCSB and TPC-C with a varying percentage of cross-partition transactions and report the results in Figure 3-10(a) and 3-10(b). When there are no cross-partition transactions, STAR has similar throughput compared with Dist. OCC and Dist. S2PL on both workloads. This is because the workload is embarrassingly parallel. Transactions do not need to hold locks for a round trip communication with asynchronous replication and epoch-based group commit. As we increase the percentage of cross-partition transactions, the throughput of PB. OCC stays almost

the same, and the throughput of other approaches drops. When 10% cross-partition transactions are present, STAR starts to outperform Dist. OCC and Dist. S2PL. For example, STAR has 2.9x higher throughput than Dist. S2PL on YCSB and 7.6x higher throughput than Dist. OCC on TPC-C. As more cross-partition transactions are present, the throughput of Dist. OCC and Dist. S2PL is significantly lower than STAR (also lower than PB. OCC), and the throughput of STAR approaches the throughput of PB. OCC. This is because STAR behaves similarly to a non-partitioned system when all transactions are cross-partition transactions.

Overall, STAR running on 4 nodes achieves up to 3x higher throughput than a primary/backup system running on 2 nodes (e.g., PB. OCC) and up to 10x higher throughput than systems employing distributed concurrency control algorithms (e.g., Dist. OCC and Dist. S2PL) on 4 nodes. As a result, we believe that STAR is a good fit for workloads with both single-partition and cross-partition transactions. It can outperform both non-partitioned and partitioning-based systems, as we envisioned in Figure 3-1.

Results of synchronous replication

We next study the performance of PB. OCC, Dist. OCC and Dist. S2PL with synchronous replication. We ran the same workload as in Section 3.6.2 with a varying percentage of cross-partition transactions and report the results in Figure 3-10(c) and 3-10(d). For clarity, the y-axis uses different scales (See Figure 3-10(a) and 3-10(b) for the results of STAR). When there are no cross-partition transactions, the workload is embarrassingly parallel. However, PB. OCC, Dist. OCC, and Dist. S2PL all have much lower throughput than STAR in this scenario. This is because even single-partition transactions need to hold locks during the round trip communication due to synchronous replication. As we increase the percentage of cross-partition transactions, we observe that the throughput of PB. OCC stays almost the same, since the throughput of a non-partitioned system is not sensitive to the percentage of cross-partition transactions in a workload. Dist. OCC and Dist. S2PL have lower throughput, since more transactions need to read from remote nodes during the ex-

Table 3.2: Latency (ms) of each approach - 50th percentile/99th percentile

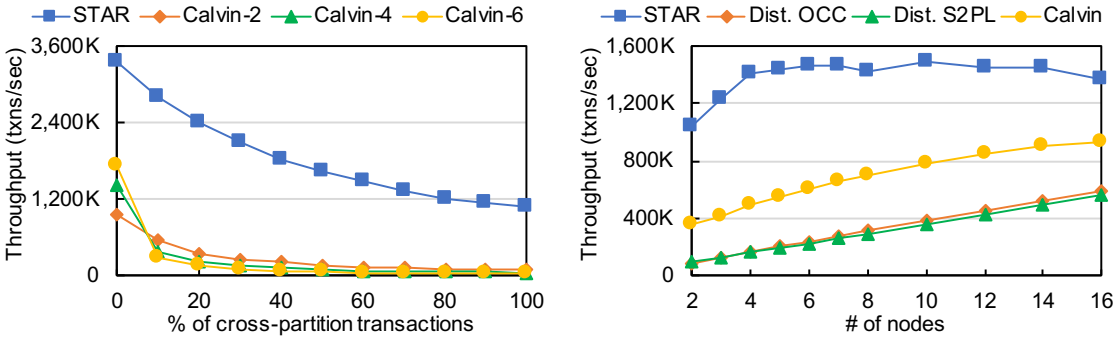
	Synchronous replication						Asynchronous replication + Epoch-based group commit
% of cross-partition transactions	YCSB			TPC-C			Each approach has a similar latency due to epoch-based group commit
	10%	50%	90%	10%	50%	90%	
STAR	-						6.2/9.4
PB. OCC	0.1/0.2	0.1/0.2	0.1/0.2	0.1/1.1	0.1/1.1	0.1/1.1	5.5/11.3
Dist. OCC	0.2/0.7	0.3/0.9	0.7/0.9	0.2/0.6	0.3/0.7	0.6/0.8	6.4/11.4
Dist. S2PL	0.2/4.5	0.4/6.0	0.6/6.7	0.2/4.9	0.5/6.8	0.8/8.9	6.2/11.2

ecution phase. They also need multiple rounds of communication to validate and commit transactions (2PC).

Overall, the throughput of PB. OCC, Dist. OCC, and Dist. S2PL is much lower than those with asynchronous replication and epoch-based group commit due to the overhead of network round trips for every transaction. STAR has much higher throughput than these approaches with synchronous replication — at least 7x higher throughput on YCSB, 15x higher throughput on TPC-C.

Latency of each approach

We now study the latency of each approach and report the latency at the 50th percentile and the 99th percentile in Table 3.2, when the percentage of cross-partition transactions is 10%, 50%, and 90%. We first discuss the latency of each approach with synchronous replication. We observe that PB. OCC's latency at the 50th percentile and the 99th percentile is not sensitive to the percentage of cross-partition transactions. Dist. OCC and Dist. S2PL have higher latency at both the 50th percentile and the 99th percentile, as we increase the percentage of cross-partition transactions. This is because there are more remote reads and the commit protocols they use need multiple round trip communication. In particular, the latency of Dist. S2PL at the 99th percentile is close to 10 ms on TPC-C. In STAR, the iteration time determines the latency of transactions. Similarly, the latency of transactions in Dist. OCC and Dist. S2PL with asynchronous replication and epoch-based group commit depends on epoch size. For this reason, STAR has similar latency at the 50th percentile and the 99th percentile to other approaches with asynchronous replication. In Table 3.2, we



(a) YCSB (b) TPC-C

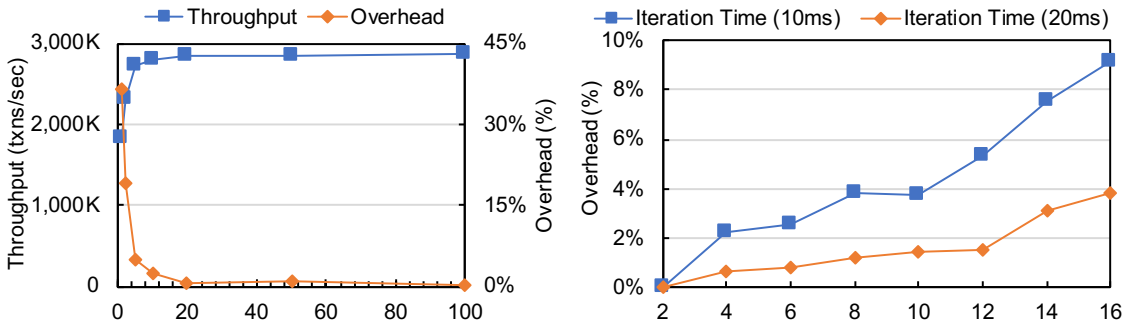
Figure 3-11: Comparison with deterministic databases

only report the results on YCSB with 10% cross-partition transactions for systems with asynchronous replication. Results on other workloads are not reported, since they are all similar to one another.

An epoch-based group commit naturally increases latency. Systems with synchronous replication have lower latency, but Figure 3-10(c) and 3-10(d) show that they have much lower throughput as well, even if no cross-partition transactions are present. In addition, the latency at the 99th percentile in systems with synchronous replication is much longer under some scenarios (e.g., Dist. S2PL on TPC-C). As prior work (e.g., Silo [98]) has argued, a few milliseconds more latency is not a problem for most transaction processing workloads, especially given throughput gains.

3.6.3 Comparison with Deterministic Databases

We next compare STAR with Calvin [97], which is a deterministic concurrency control and replication algorithm. In Calvin, a central sequencer determines the order for a batch of transactions before they start execution. The transactions are then sent to all replica groups of the database to execute deterministically. In Calvin, a replica group is a set of nodes containing a replica of the database. All replica groups will produce the same results for the same batch of transactions due to determinism. As a result, Calvin does not perform replication at the end of each transaction. Instead, it replicates inputs at the beginning of the batch of transactions and deterministically



(a) Iteration time (ms)

(b) # of nodes

Figure 3-12: Overhead of phase transitions

executes the batch across replica groups.

We implemented the Calvin algorithm in C++ in our framework as well to have a fair comparison. The original design of Calvin uses a single-threaded lock manager to grant locks to multiple execution threads following the deterministic order. To better utilize more CPU resources, we implemented a multi-threaded lock manager, in which each thread is responsible for granting locks in a different portion of the database. The remaining CPU cores are used as worker threads to execute transactions.

Increasing the number of threads for the lock manager does not always improve performance. The reasons are twofold: (1) fewer threads are left for transaction execution (2) more communication is needed among worker threads for cross-partition transactions. In this experiment, we consider three configurations with different number of threads used in the lock manager in Calvin, namely (1) Calvin-2, (2) Calvin-4 and (3) Calvin-6. We use Calvin-x to denote the number of threads used for the lock manager, i.e, there are $12 - x$ threads executing transactions. In all configurations, we study the performance of Calvin in one replica group on 4 nodes. The results of Calvin-1 and Calvin-3 are not reported, since they never deliver the best performance.

We report the results on YCSB and TPC-C with a varying percentage of cross-partition transactions in Figure 3-11(a) and 3-11(b). When there are no cross-partition transactions, Calvin-6 achieves the best performance, since more parallelism is exploited (i.e., 6 worker threads on each node, yielding a total of 24 worker threads). Calvin-2 and Calvin-4 have lower throughput as the worker threads are not saturated

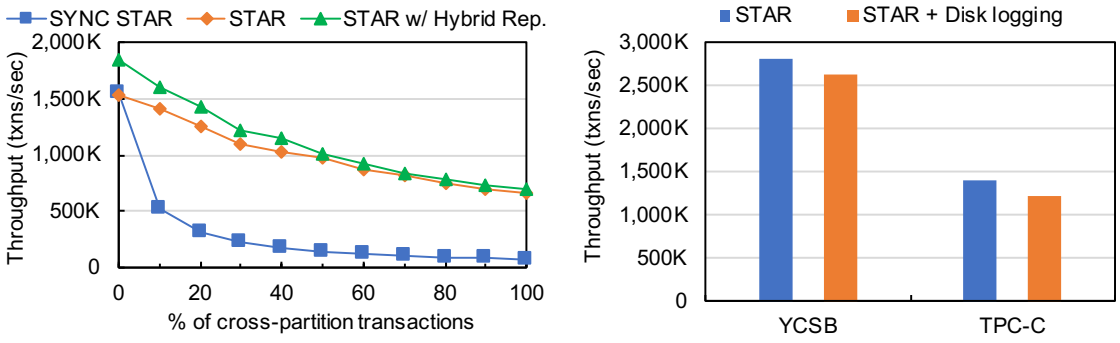
when fewer threads are used for the lock manager. In contrast, STAR uses 12 worker threads on each node, yielding a total of 48 worker threads and has 1.4-1.9x higher throughput than Calvin-6. When all transactions are cross-partition transactions, Calvin-2 has the best performance. This is because Calvin-4 and Calvin-6 needs more synchronization and communication. Overall, STAR has 4-11x higher throughput than Calvin with various configurations.

3.6.4 The Overhead of Phase Transitions

We now study how the iteration time of a phase switch affects the overall throughput of STAR and the overhead due to this phase switching algorithm with a YCSB workload. Similar results were obtained on other workloads but are not reported due to space limitations. We varied the iteration time of the phase switching algorithm from 1 ms to 100 ms and report the system's throughput and overhead in Figure 3-12(a). The overhead is measured as the system's performance degradation compared to the one running with a 200 ms iteration time. Increasing the iteration time decreases the overhead of the phase switching algorithm as expected, since less time is spent during the synchronization. For example, when the iteration time is 1 ms, the overhead is as high as 43% and system only achieves around half of its maximum throughput (i.e., the throughput achieved with 200 ms iteration time). As we increase the iteration time, the system's throughput goes up. The throughput levels off when the iteration time is larger than 10 ms. On a cluster of 4 nodes, the overhead is about 2% with a 10 ms iteration time.

We also study the overhead of phase transitions with a varying number of nodes. We ran the same YCSB workload and report the results of 10 ms and 20 ms iteration time in Figure 3-12(b). Note that we also scale the number of partitions in the database correspondingly. For example, on a cluster of 16 nodes, there are $16 \times 12 = 192$ partitions in the database. In general, the overhead of phase transitions is larger with more nodes on a cluster due to variance of communication delays. In addition, a shorter iteration time makes the overhead smaller (20 ms vs. 10 ms).

Overall, the overhead of phase transitions is less than 5% with a 10 ms iteration



(a) Replication strategies

(b) Fault tolerance

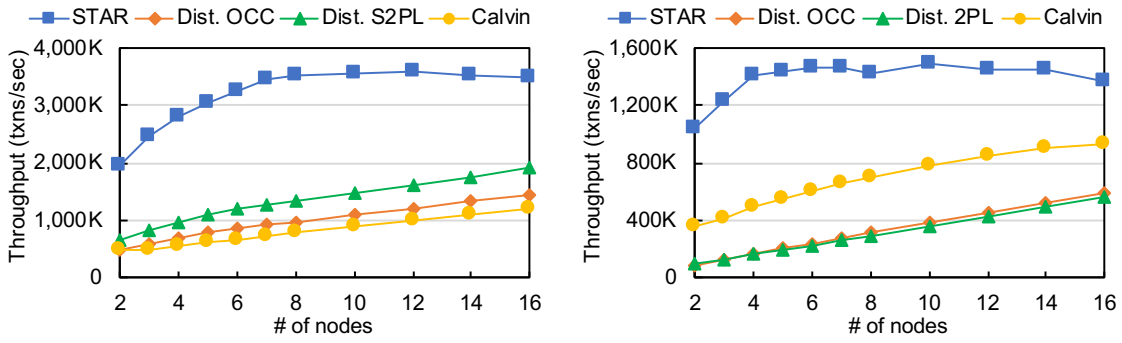
Figure 3-13: Replication and fault tolerance experiment

time on a cluster of less than 10 nodes. In all experiments in this chapter, we set the iteration time to 10 ms. With this setting, the system can achieve more than 95% of its maximum throughput and have good balance between throughput and latency.

3.6.5 Replication and Fault Tolerance

We now study the effectiveness of STAR's asynchronous replication in the single-master phase and the effectiveness of hybrid replication. We only report the results from TPC-C in this experiment, since a transaction in YCSB always updates the whole record. In Figure 3-13(a), `SYNC STAR` shows the performance of STAR that uses synchronous replication in the single-master phase. `STAR` indicates the one with asynchronous replication. `STAR w/ Hybrid Rep.` further enables operation replication in the partitioned phase on top of `STAR`. When there are more cross-partition transactions, `SYNC STAR` has much lower throughput than `STAR`. This is because more network round trips are needed during replication in the single-master phase. The improvement of `STAR w/ Hybrid Rep.` is also less significant, since fewer transactions are run in the partitioned phase.

We next show the performance degradation of STAR when disk logging is enabled. We ran both YCSB and TPC-C workloads and report the results in Figure 3-13(b). In summary, the overhead of disk logging and checkpointing is 6% in YCSB and 14% in TPC-C. Note that non-partitioned and partitioning-based systems would experience



(a) YCSB (b) TPC-C

Figure 3-14: Scalability experiment

similar overheads from disk logging.

3.6.6 Scalability Experiment

In this experiment, we study the scalability of STAR on both YCSB and TPC-C. We ran the experiment with a varying number of `m5.4xlarge` nodes and report the results in Figure 3-14. Note that the database is scaled correspondingly as we did in Section 3.6.4. On YCSB, STAR with 8 nodes achieves 1.8x higher throughput than STAR with 2 nodes. The performance of STAR stays stable beyond 8 nodes. On TPC-C, STAR with 4 nodes achieves 1.4x higher throughput than STAR with 2 nodes. The system stops scaling with more than 4 nodes. This is because the system saturates the network with 4 nodes (roughly 4.8 Gbits/sec). In contrast, Dist. OCC, Dist. S2PL and Calvin start with lower performance but all have almost linear scalability.

We believe it is possible for distributed partitioning-based systems to have competitive performance to STAR, although such systems will likely require more nodes to achieve comparable performance. Assuming linear scalability and the network not becoming a bottleneck (the ideal case for baselines), distributed partitioning-based systems maybe outperform STAR on YCSB and TPC-C with roughly 30-40 nodes.

3.7 Related Work

STAR builds on a number of pieces of related work for its design, including in-memory transaction processing, replication and durability.

3.7.1 In-memory Transaction Processing

Modern fast in-memory databases have long been an active area of research [21, 30, 52, 72, 91, 98, 108]. In H-Store [91], transactions local to a single partition are executed by a single thread associated with the partition. This extreme form of partitioning makes single-partition transactions very fast but creates significant contention for cross-partition transactions, where whole-partition locks are held. Silo [98] divides time into a series of short epochs and each thread generates its own timestamp by embedding the global epoch to avoid shared-memory writes, avoiding contention for a global critical section. Because of its high throughput and simple design, we adopted the Silo architecture for STAR, reimplementing it and adding our phase-switching protocol and replication. Doppel [70] executes highly contentious transactions in a separate phase from other regular transactions such that special optimizations (i.e., commutativity) can be applied to improve scalability.

F1 [89] is an OCC protocol built on top of Google's Spanner [19]. MaaT [59] reduces transaction conflicts with dynamic timestamp ranges. ROCOCO [67] tracks conflicting constituent pieces of transactions and re-orders them in a serializable order before execution. To reduce the conflicts of distributed transactions, STAR runs all cross-partition transactions on a single machine in the single-master phase. Clay [86] improves data locality to reduce the number of distributed transactions in a distributed OLTP system by smartly partitioning and migrating data across the servers. Some previous work [55, 23, 15] proposed to move the master node of a tuple dynamically, in order to convert distributed transactions into local transactions. Unlike STAR, however, moving the mastership still requires network communication. FaRM [31], FaSST [44] and DrTM [105] improve the performance of a distributed OLTP database by exploiting RDMA. STAR can use RDMA to further decrease the

overhead of replication and the phase switching algorithm as well.

3.7.2 Replicated Systems

Replication is the way in which database systems achieve high availability. Synchronous replication was popularized by systems like Postgres-R [46] and Galera Cluster [38], which showed how to make synchronous replication practical using group communication and deferred propagation of writes. Tashkent [33] is a fully replicated database in which transactions run locally on a replica. To keep replicas consistent, each replica does not communicate with each other but communicates to a certifier, which decides a global order for update transactions. Calvin [97] replicates transactions requests among replica groups and assigns a global order to each transaction for deterministic execution [96], allowing it to eliminate expensive distributed coordination. However, cross-node communication is still necessary during transaction execution because of remote reads. Mencius [63] is a state machine replication method that improves Paxos to achieve high throughput under high client load and low latency under low client load by partitioning sequence numbers, even under changing wide-area network environments. HRDB [99] tolerates Byzantine faults among replicas by scheduling transactions with a commit barrier. Ganymed [75, 76] runs update transactions on a single node and runs read-only transactions on a potentially unlimited number of replicas, allowing the system to scale read-intensive workloads. STAR is the first system that dynamically changes the mastership of records, to avoid distributed coordination. Neither a global order nor group communication is necessary, even for cross-partition transactions, since we run these cross-partition transactions in parallel on a single node.

3.7.3 Recoverable Systems

H-Store [61] uses transaction-level logging. It periodically checkpoints a transactionally consistent snapshot to disk and logs all the parameters of stored procedures. H-Store executes transactions following a global order and replays all the transac-

tions in the same order during recovery. SiloR [113] uses a multi-threaded parallel value logging scheme that supports parallel replay in non-partitioned databases. In contrast, transaction-level logging requires that transactions be replayed in the same order. In STAR, different replication strategies, including both SiloR-like parallel value replication and H-Store-like operation replication are used in different phases, significantly reducing bandwidth requirements.

3.8 Summary

In this chapter, we presented STAR, a new distributed in-memory database with asymmetric replication. STAR employs a new *phase-switching* scheme where single-partition transactions are run on multiple machines in parallel, and cross-partition transactions are run on a single machine by re-mastering records on the fly, allowing us to avoid cross-node communication and the use of distributed commit protocols like 2PC for distributed transactions. Our results on YCSB and TPC-C show that STAR is able to dramatically exceed the performance of systems that employ conventional concurrency control and replication algorithms by up to one order of magnitude.

Chapter 4

COCO: Epoch-based Commit and Replication in Distributed OLTP Databases

4.1 Introduction

In this chapter, we make the key observation that the inefficiency of both 2PC and synchronous replication mainly comes from the fact that existing protocols enforce consistency at the granularity of *individual transactions*. By grouping transactions that arrive within a short time window into short periods of time — which we call epochs — it's possible to manage both atomic commit and consistent data replication at the granularity of epochs. In our approach, an epoch is the basic unit at which transactions commit and recover — either all or none of the transactions in an epoch commit — which adheres to the general principle of group commit [26] in single-node databases. However, epoch-based commit and replication focus on reducing the overhead due to 2PC and synchronous replication rather than disk latency as in group commit. As a result, a transaction releases its locks immediately after execution finishes, and logging to persistent storage occurs in the background and is only enforced at the boundary of epochs. Similarly, a transaction no longer needs to

hold locks after updating the primary replica, since write propagation happens in the background as in asynchronous replication. Note that the writes of a transaction are visible to other transactions as soon as it commits but they may disappear if a failure occurs and a rollback happens. Therefore, a transaction does not release the result to the end user until the current epoch commits, when the writes of all transactions belong to the epoch are durable. In COCO, consistency is enforced at the boundary of epochs as in synchronous replication. The epoch size, which determines the average latency of transactions, can be selected to be sufficiently low for most OLTP workloads (e.g., 10 ms). Prior work has argued that such latencies are acceptable [21, 70, 98]. In addition, epoch-based commit and replication can help reduce tail latency compared to a traditional architecture with 2PC [30].

In this chapter, we describe COCO, a distributed main-memory OLTP database we built that embodies the idea of epoch-based commit and replication. COCO supports two variants of optimistic concurrency control (OCC) [49] that serialize transactions using physical time and logical time, respectively. In addition, it supports both serializability and snapshot isolation with various optimizations, which are enabled by the epoch-based commit and replication.

4.2 Epoch-based Commit and Replication

In this section, we first show how our new commit protocol based on epochs offers superior performance and provides the same guarantees as two-phase commit (2PC). We then discuss how the epoch-based design of COCO reveals opportunities to design a new replication scheme that unifies the best of both synchronous and asynchronous replication schemes. In Section 4.3 and Section 4.4, we will discuss how to design distributed concurrency control with epoch-based commit and replication in COCO.

4.2.1 The Commit Protocol

In COCO, a batch of transactions run and commit in an epoch. However, the result of each transaction is not released until the end of the epoch, when all participant nodes

agree to commit all transactions from the current epoch. The system increments the global epoch every few milliseconds (e.g., 10 ms by default) with two phases: a *prepare* phase and a *commit* phase, as in two-phase commit (2PC).

In the prepare phase, the coordinator sends a prepare message to each participant node. Note that the coordinator node is the node coordinating epoch advancement among a cluster of nodes and it's different from the coordinator of distributed transactions as we will see in later sections. The coordinator can be any node in the system or a standalone node outside the system. To prevent the coordinator from being a single point of failure, it can be implemented as a replicated state machine with Paxos [50] or Raft [73]. When a participant node receives a prepare message, it prepares to commit all transactions in the current epoch by force logging a durable *prepared write* record (indicated by a purple star in Figure 4-1) with all the transaction IDs (TIDs) of ready-to-commit transactions as well as the current epoch number. Note that some transactions may have aborted earlier due to conflicts. The underlying concurrency control algorithms are also required to log all the writes of ready-to-commit transactions durably (See Section 4.5.2) before the prepared write record. When a participant node durably logs all necessary writes, it then replies an acknowledgement to the coordinator.

In the commit phase, the coordinator first decides if the current epoch can commit. If any participant node fails to reply an acknowledgement due to failures, all transactions from the current epoch will abort. Otherwise, the coordinator writes a durable *commit* record (indicated by a purple star in Figure 4-1) with the current epoch number and then increments the global epoch. It then sends a commit message to each participant node. Note that if a transaction aborts due to concurrent accesses or integrity violation, it does not stop the current epoch to commit. When a participant node receives a commit message, all the writes of ready-to-commit transactions from the last epoch are considered committed, and the results of these transactions are released to users. In the end, it replies an acknowledgement to the coordinator, and prepares to execute transactions from the next epoch.

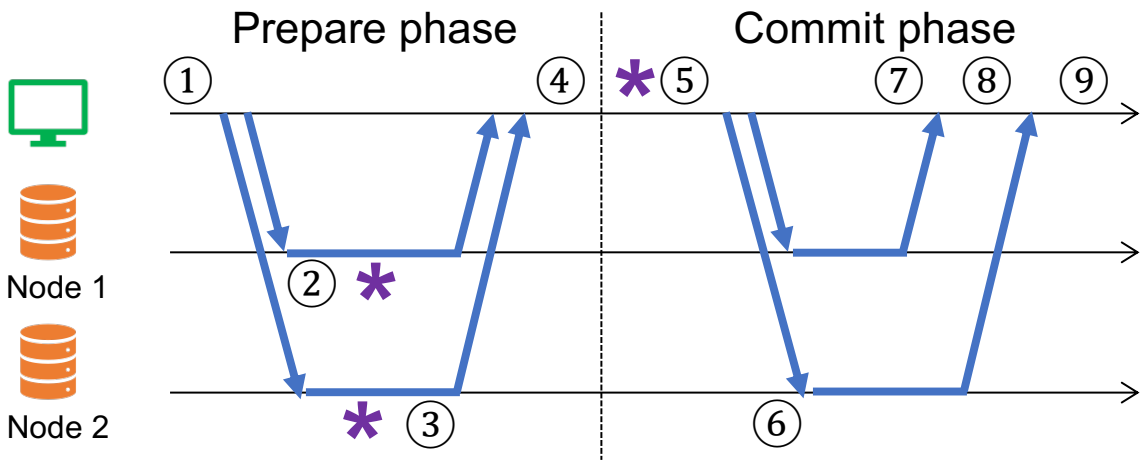


Figure 4-1: Failure scenarios in epoch-based commit

4.2.2 Fault Tolerance

In this thesis, we assume fail-stop failures [25], in which we can assume that any healthy node in the system can detect which node has failed. A *failed node* is one on which the process of a COCO instance has crashed. Since COCO detects failures at the granularity of epochs, all transactions in an epoch will be aborted and re-executed by the system automatically when a failure occurs. Here, we argue that the benefit brought by COCO significantly exceeds the cost to abort and re-run a whole epoch of transactions, since failures are rare on modern hardware. 2PC ensures atomicity and durability at the granularity of every single transaction, but introduces expensive coordination, which is wasted most of the time.

As shown in Figure 4-1, a failure can occur when an epoch of transactions commit with the epoch-based commit protocol. We classify all failure scenarios into nine categories: (1) before the coordinator sends prepare requests, (2) after some participant nodes receive prepared requests, (3) after all participant nodes receive prepared requests, (4) before the coordinator receives all votes from participant nodes, (5) after the coordinator writes the commit record, (6) before some participant nodes receive commit requests, (7) before the coordinator receives any acknowledgement, (8) after the coordinator receives some acknowledgements, and (9) after the coordinator receives all acknowledgements.

The durable commit record written on the coordinator node indicates the time when the system commits all transactions from the current epoch. Therefore, in cases (1) - (4), the system simply aborts all transactions from the current epoch. Specifically, after recovery, each participant node rollbacks its prepared writes and discards all intermediate results. In cases (5) - (8), transactions from the current epoch are considered committed even though a failure has occurred. After recovery, each participant node can learn the outcome of the current epoch when communicating with the coordinator, and then releases the results to users. Case (9) is the same as case (1), since the system has entered the next epoch.

Once a fault occurs, COCO rollbacks the database to the last successful epoch, i.e., all tuples that are updated in the current epoch are reverted to the states in the last epoch. To achieve this, the database maintains two versions of each tuple. One always has the latest value. The other one has the most recent value up to the last successful epoch.

4.2.3 Efficient and Consistent Replication

A desirable property of any approach to high availability is strong consistency between replicas, i.e., that there is no way for clients to tell when a failover happened, because the state reflected by the replicas is identical. Enforcing strong consistency in a replicated and distributed database is a challenging task. The most common approach to achieve strong consistency is based on primary-backup replication, where the primary releases locks and commits only after writes have propagated to all replicas, blocking other transactions from accessing modified records and limiting performance.

If a transaction could process reads at replicas and asynchronously ship writes to replicas, it could achieve considerably lower latency and higher throughput, because transactions can read from the nearest replica and release locks before replicas respond to writes. Indeed, these features are central to many recent systems that offer eventual consistency(e.g., Dynamo [25]). Observe that both local reads and asynchronous writes introduce the same problem: the possibility of stale reads at replicas. Thus, they both introduce the same consistency challenge: the database cannot de-

termine whether the records a transaction reads are consistent or not. Therefore, transactions running with two-phase locking (2PL) [10, 34] must always read from the primary replica, since there is no way for them to tell if records are locked by only communicating with backup replicas.

In COCO, atomicity and durability are enforced at the granularity of epochs. Therefore, COCO can replicate writes of committed transactions asynchronously without having to worry about the replication lag within an epoch. The system only needs to ensure that the primary replica is consistent with all backup replicas at epoch boundaries. In addition, COCO uses optimistic concurrency control and each record in the database has an associated TID. The TID of a record usually indicates the last transaction that modified the record and can be used to detect if a read from backup replicas is stale [58, 98, 107, 110]. As a result, a transaction in COCO can read from nearest backup replicas and only validates with the primary replica in the commit phase, which significantly reduces network traffic and latency.

4.2.4 Discussion

COCO offers superior performance to distributed transactional databases by running transactions in epochs. We now discuss the implications of epoch-based commit and replication, and how they impact existing OLTP applications. First, the epoch-based commit and replication add a few milliseconds more latency to every single transaction. As prior work (e.g., Silo [98]) has argued, this is not a problem for most transactional workloads, unless the workloads require extremely low latency. In addition, running transactions with epochs helps reduce tail latency as well [30]. Second, the system could have undesirable performance due to imbalanced running time among transactions. For example, a single long-running transaction can stop a whole batch of transactions from committing until it finishes. In reality, most update transactions are short-lived and most of long-running transactions are read-only analytical transactions, which can be configured to run over a slightly stale database snapshot. Third, since failures are detected at the granularity of epochs, all transactions in an epoch will be aborted and re-executed by the system automatically

```
1  Function: transaction_read(T, key)
2      ns = get_replica_nodes(key)
3      if node_id() in ns: # a local copy is available
4          record = db_read(key)
5      else: # n is the nearest node chosen from ns
6          record = call_n(db_read, key)
7      T.RS.push_back(record)
8
9  Function: db_read(key)
10     # atomically load value and TID
11     return db[tuple.key].{value, tid}
```

Figure 4-2: Pseudocode to read from the database

when a failure occurs. Here, we argue that the benefit brought by epoch-based commit and replication significantly exceeds the cost to abort and re-run a whole epoch of transactions, since failures are rare on modern hardware. Note that a transaction, which aborts due to conflicts, does not affect the commit/abort decision of a whole epoch and it will be re-run by the system automatically.

4.3 The Lifecycle of a Transaction in COCO

In this section, we discuss the lifecycle of a distributed transaction in COCO, which contains an execution phase and a commit phase.

4.3.1 The Execution Phase

A transaction in COCO runs in two phases: an execution phase and a commit phase. We say the node initiating a transaction is the *coordinator node*, and other nodes are *participant nodes*.

In the execution phase, a transaction reads records from the database and maintains local copies of them in its *read set* (RS). Each entry in the read set contains the value as well as the record's associated transaction ID (TID). For a read request, the

coordinator node first checks if the request's primary key is already in the read set. This happens when a transaction reads a data record multiple times. In this case, the coordinator node simply uses the value of the first read. Otherwise, the coordinator node reads the record from the database.

A record can be read from any replica in COCO. To avoid network communication, the coordinator node always reads from its local database if a local copy is available. As shown in Figure 4-2, the coordinator first locates the nodes ns on which there exists a copy of the record. If the coordinator node happens to be from ns, a transaction can simply read the record from its local database. If no local copy is available, a read request is sent to the nearest node n chosen from ns. In COCO, TIDs are associated with records at both primary and backup replicas. For a read request, the system returns both the value and the TID of a record; and both are stored in the transaction's local read set.

All computation is performed in the execution phase. Since COCO's algorithm is optimistic, writes are not applied to the database but are stored in a per-transaction *write set* (WS), in which, as with the read set, each entry has a value and the record's associated TID. For a write operation, if the primary key is not in the write set, a new entry is created with the value and then inserted into the write set. Otherwise, the system simply updates the write set with the new value. Note that for updates to records that are already in the read set, the transaction also copies the TIDs to the entry in the write set, which are used for validation later on.

4.3.2 The Commit Phase

After a transaction finishes its execution phase, it must be successfully validated before it commits. We now describe the three steps to commit a transaction: (1) lock all records in the transaction's write set; (2) validate all records in the transaction's read set and generate a TID; (3) commit changes to the database. In Section 4.4, we will discuss the details of two concurrency control algorithms.

Locking the write set

A transaction first tries to acquire locks on each record in the write set to prevent concurrent updates from other transactions. In COCO, a locking request is only sent to the primary replica of each record. To avoid deadlocks, we adopt a `NO_WAIT` deadlock prevention policy, which was shown as the most scalable protocol [42]. In `NO_WAIT`, if the lock is already held on the record, the transaction does not wait but simply aborts. For each acquired lock, if any record's latest TID does not equal to the stored TID, the transaction aborts as well. This is because the record has been changed at the primary replica since the transaction last read it.

Validating the read set & TID assignment

When a transaction has locked each record in its write set, it begins to validate each record in its read set. Unlike the execution phase, in which a read request is sent to the nearest node with a copy of the record, a read validation request is always sent to the primary replica of each record. A transaction may fail to validate a record due to concurrent transactions that access the same record. For example, a record cannot be validated if it is being locked by another transaction or the value has changed since its last read.

COCO assigns a TID to each transaction as well in this step. The assignment can happen either prior to or after read validation [58, 98, 107, 110, 108, 109], depending on whether the TID is used during read validation. There are some conditions to assign a TID. For example, it must be able to tell the order of conflicting transactions. We now assume a TID is correctly assigned and leave the details of TID assignment to Section 4.4.

Writing back to the database

If a transaction fails the validation, it simply aborts, unlocks the acquired locks, and discards its local write set. Otherwise, it will commit changes in its write set to the database. COCO applies the writes and replication asynchronously to reduce

```
1  Function: transaction_write(T)
2      for record in T.WS:
3          n = get_primary_node(record.key)
4          call_n(db_write, record.key, record.value, T.tid)
5          for n in get_replica_nodes(record.key) \ {n}:
6              call_n(db_replicate, record.key, record.value, T.tid)
7
8  Function: db_write(key, value, tid)
9      db[key] = {value, tid}
10     unlock(db[key])
11
12 Function: db_replicate(key, value, tid)
13     # begin atomic section
14     if db[key].tid < tid: # Thomas write rule
15         db[key] = {value, tid}
16     # end atomic section
```

Figure 4-3: Pseudocode to write to the database

round-trip communication. Other transactions can observe the writes of committed transactions at each replica as soon as a write is written to the database. Note that, to ensure the writes of committed transactions durable across failures, the result of a committed transaction is not released to clients until the end of the current epoch.

As illustrated in Figure 4-3, the value of each record in a transaction's write set and the generated TID are sent to the primary and backup replicas from the coordinator node. There are two scenarios that a write is applied to the database: (1) the write is at the primary replica: Since the primary replica is holding the lock, upon receiving the write request, the primary replica simply updates the value and the TID, and then unlocks the record; (2) the write is at a backup replica: Since asynchronous replication is employed in COCO, upon receiving the write request, the lock on the record is not necessarily held on the primary replica, meaning replication requests to the same record from multiple transactions could arrive out of order. COCO determines whether a write at a backup replica should be applied using the

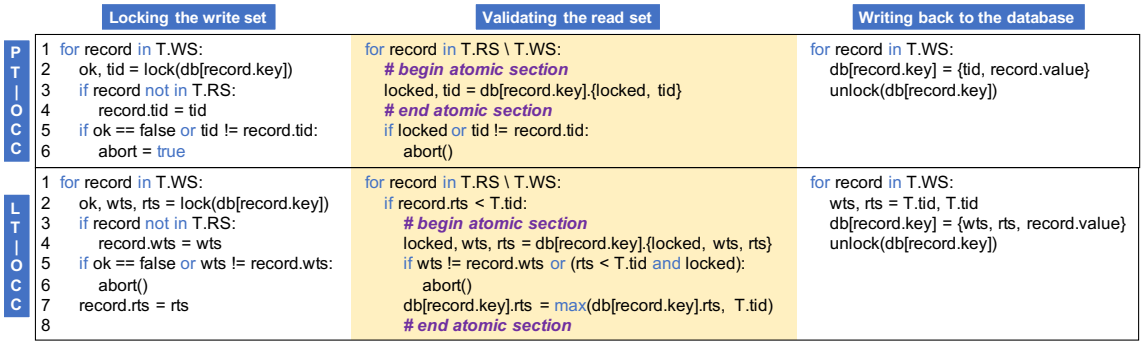


Figure 4-4: Pseudocode of the commit phase in PT-OCC and LT-OCC

Thomas write rule [95]: the database only applies a write at a backup replica if the record's current TID is less than the TID associated with the write (line 14 – 15 of Figure 4-3). Because the TID of a record monotonically increases at the primary replica, this guarantees that backup replicas apply the writes in the same order as the order to commit transactions at the primary replica.

4.4 Concurrency Control in COCO: the Two Variants of OCC

In this section, we first describe how to design distributed concurrency control with epoch-based group commit in COCO. In particular, we discuss how to adapt two popular singe-node concurrency control algorithms (i.e., Silo [98] and Tictoc [108]) into COCO. We next discuss the tradeoffs between these two concurrency control algorithms in the distributed environment and the extension to snapshot transactions.

4.4.1 PT-OCC – Physical Time OCC

Many recent single-node concurrency control protocols [58, 70, 110, 113] adopted the design philosophy of Silo [98], in which "anti-dependencies" (i.e., write-after-read conflicts) are not tracked to avoid scaling bottlenecks. Instead, anti-dependencies are only enforced across epochs boundaries, which naturally fits into the design of COCO based on epoch-based commit and replication.

We now discuss how to adapt Silo to a distributed environment in COCO and present the pseudocode on the top of Figure 4-4. A transaction first locks each record in its write set. If any record is locked by another transaction, the transaction simply aborts. The transaction next validates the records that only appear in its read set. The validation would fail in two scenarios: (1) the record's TID has changed, meaning the record was modified by other concurrent transactions; (2) the record is locked by another transaction. In either case, the transaction must abort, unlock the acquired locks, and discard its local write set. If the transaction successfully validates its read set, the TID is next generated. There are three criteria [98] to generate the TID for each transaction in PT-OCC: (1) it must be in the current global epoch; (2) it must be larger than the TID of any record in the read/write set; (3) it must be larger than the worker thread's last chosen TID. At last, the transaction commits changes in its write set to the database. The value of each record in the transaction's write set and the TID are sent to the primary replica. As discussed in Section 4.3, the writes are also asynchronously replicated to backup replicas from the coordinator node.

The protocol above guarantees serializability because all written records have been locked before validating the TIDs of read records. A more formal proof of correctness through reduction to strict two-phase locking can be found in Silo [98].

4.4.2 LT-OCC – Logical Time OCC

Many concurrency control algorithms [52, 80, 106] allow read-only snapshot transactions to run over a consistent snapshot of the database and commit back in time. TicToc [108], a single-node concurrency control algorithm, takes a further step by allowing read-write serializable transactions to commit back in time in the space of logical time. In TicToc, each record in the database is associated with two logical timestamps, which are represented by two 64-bit integers: $[\texttt{wts}, \texttt{rts}]$. The `wts` is the logical write timestamp, indicating when the record was written, and the `rts` is the logical *read validity* timestamp, indicating that the record can be read at any logical time ts such that $\texttt{wts} \leq ts \leq \texttt{rts}$. The key idea is to dynamically assign a logical timestamp (i.e., TID) to each transaction on commit so that each record in

the read/write set is available for read and update at the same logical time.

We now discuss how to adapt TicToc [108] to a distributed environment in COCO and present the pseudocode on the bottom of Figure 4-4. A transaction first locks each record in its write set. Since concurrent transactions may have extended the `rts` of some records, LT-OCC updates the `rts` of each record in the transaction's write set to the latest one at the primary replica, which is available when a record is locked. The transaction next validates the records that only appear in its read set. The validation requires a TID, which is the smallest timestamp that meets the following three conditions [110]: (1) it must be in the current global epoch; (2) it must be not less than the `wts` of any record in the read set; (3) it must be larger than the `rts` of any record in the write set. The TID is first compared with the `rts` of the record in its read set. A read validation request is sent only when a record's `rts` is less than the TID. In this case, the transaction tries to extend the record's `rts` at the primary replica. The extension would fail in two scenarios: (1) the record's `wts` has changed, meaning the record was modified by other concurrent transactions; (2) the record is locked by another transaction and the `rts` is less than the TID. In either case, the `rts` cannot be extended and the transaction must abort. Otherwise, the transaction extends the record's `rts` to the TID. If the transaction fails the validation, it simply aborts, unlocks the acquired locks, and discards its local write set. Otherwise, it will commit changes in its write set to the database. As in PT-OCC, the writes are also asynchronously replicated to backup replicas from the coordinator node.

LT-OCC is able to avoid the need to validate the read set against the primary replica as long as the logical commit time falls in all time intervals of data read from replicas, even if the replicas are not fully up to date. Informally, the protocol above guarantees serializability because all the reads and writes of a transaction happen at the same logical time. From logical time perspective, all accesses happen simultaneously. A more formal proof of how logical timestamps enforce serializability can be found in TicToc [108].

4.4.3 Tradeoffs in PT-OCC and LT-OCC

In PT-OCC, TIDs are always monotonically increasing between conflicting transactions. To achieve this, the database validates a transaction's read set by comparing the data versions from the read set to the latest ones at the primary replica. If any record's primary partition is not on the coordinator, a round-trip communication must be performed when a transaction's read set is validated. In contrast, LT-OCC achieves less coordination in transaction validation. For example, for each record in a transaction's read set, the system first compares the record's `rts` and the transaction's assigned TID. A read validation request is sent only when the record's `rts` is less than the TID and the primary node of the record is not the coordinator node of the transaction. Otherwise, the read is already consistent, since it is valid at the TID's logical time. If all records in the read set can be validated locally, a round trip communication is eliminated entirely.

The cost of reduced read validation is that LT-OCC sacrifices external consistency [19], i.e., the system commits transactions under non order-preserving serializability, which we will describe below. We now use an example to show the events happening following the physical time in Figure 4-5. Consider the example in the left side of Figure 4-5, in which transaction T_1 and T_2 run concurrently. By the time T_1 tries to commit, T_2 has committed and a new value of record y has been written to the database with version at time 11, which is the smallest TID larger than the TID of any record in T_2's read and write set. Since the TID of record y in T_1's read set has changed from 10 to 11, T_1 cannot commit at time 11 and must abort. T_1 must retry, reads the new value of record y, and commits at time 12. In systems with order-preserving serializability (e.g., Spanner [19]), the commit time of conflicting transactions determines transaction commit order.

In contrast, transactions commit in LT-OCC do not necessarily follow the order of commit timestamps, i.e., the logical time does not always agree to the physical time. Consider the example in the right side of Figure 4-5. After transaction T_2 commits, which has written a new value of record y: $\big[\texttt{wts} = 21, \texttt{rts} = 21\big]$. Transaction T_1

can still commit at time 16, even though record y is in its read set and the value has changed. This is because T_1's commit time is earlier than record y's last written time in the space of logical time, i.e., 16 falls between logical time 10 and 20.

LT-OCC reduces round trip communication during transaction validation. The performance advantage is even more significant in the Wide-Area Network (WAN) setting. However, it may fail to meet the requirement of some applications. For example, a user may want to see his/her own photo after uploading it. In this scenario, LT-OCC may miss the update if a transaction reads from a backup replica, which has not received the write in time (i.e., upload of a photo). For applications that cannot miss updates, COCO allows them to always read from the primary replica at the cost of more expensive communication.

4.4.4 Snapshot Transactions

Serializability allows transactions to run concurrently while ensuring the state of the database is equivalent to some serial ordering of the transactions. In contrast, snapshot transactions only run over a consistent snapshot of the database, meaning read/write conflicts are not detected. As a result, a database system running snapshot isolation has a lower abort rate and higher throughput.

Many systems adopt a multi-version concurrency control (MVCC) algorithm to support snapshot isolation (SI). In an MVCC-based system, a timestamp is assigned to a transaction when it starts to execute. By reading all records that have overlapping time intervals with the timestamp, the transaction is guaranteed to observe the state of the database (i.e., a consistent snapshot) at the time when the transaction began. Instead of maintaining multiple versions for each record, we made minor changes to the algorithm discussed in Section 4.4 to support snapshot isolation. The key idea behind is to ensure that all reads are from a consistent snapshot of the database and there are no conflicts with any concurrent updates made since that snapshot. We now describe the extensions making both PT-OCC and LT-OCC support snapshot transactions.

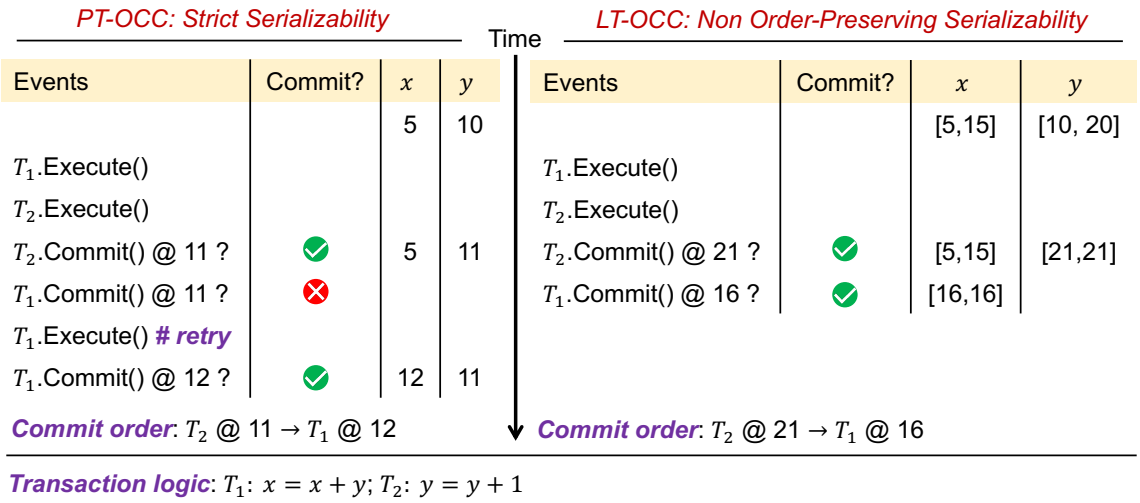


Figure 4-5: Illustrating strict serializability vs. non order-preserving serializability

Snapshot transactions in PT-OCC

In PT-OCC, a transaction first locks all records in the write set and next validates each record in the read set to see if there exists any concurrent modification. If the validation succeeds, the transaction can safely commit under serializability. There is a short period of time after all records in the write set are locked and before any record in the read set is validated. The serialization point can be any physical time during the period.

To support snapshot transactions, a transaction can validate its read set without locks being held for each record in its write set. As long as no changes are detected, it is guaranteed that the transaction reads a consistent snapshot from the database. Likewise, there is a short period of time after the transaction finishes all reads and before the validation, and the snapshot is taken at some physical time during the period above. Snapshot transactions are more likely to commit than serializable transactions, since read validation can happen right after execution without locks being held.

Snapshot transactions in LT-OCC

In LT-OCC, the TID is assigned to each serializable transaction after all records in the write set have been locked. The TID must be larger than the `rts` of each record in the write set and not less than the `wts` of each record in the read set. To support snapshot transactions, LT-OCC assigns an additional TID_{SI} to each transaction, which is the smallest timestamp not less than the `wts` of each record in the read set. Since the new TID_{SI} is not required to be larger than the `rts` of each record in the write set, it's usually less than the TID for serializable transactions allowing more transactions to commit. During read validation, a transaction can safely commit under snapshot isolation as long as each record in the read set does not change until logical time TID_{SI}.

Parallel locking and validation optimization

As we discussed in sections above, a snapshot transaction can validate its read set regardless of its write set in both PT-OCC and LT-OCC. In particular, read validation can happen before the records in the write set have been locked in PT-OCC. Likewise, the calculation of TID_{SI} does not depend on the `rts` of each record in the write set in LT-OCC, which indirectly implies that read validation can happen independently.

We now introduce an optimization we call *parallel locking and validation optimization*, which combines the first two steps in the commit phase. In order words, locking the write set and validating the read set are now allowed to happen at the same time, making both PT-OCC and LT-OCC have one fewer round trip of network communication and higher throughput.

Concurrency anomaly detection in LT-OCC

In practice, database transactions are often executed under reduced isolation levels, as there is an inherent trade-off between performance and isolation levels. For example, both Oracle and Microsoft SQL Server default to read committed. Unfortunately, such weaker isolation levels can result in concurrency anomalies, which yield

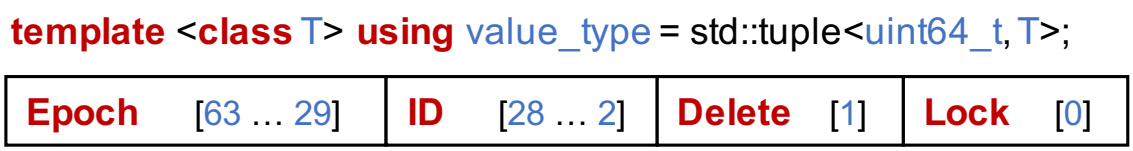


Figure 4-6: The TID format in PT-OCC

an interleaving of operations that could not arise in a serial execution of transactions.

COCO provides a real-time breakdown of how many transactions may have experienced anomalies when running under snapshot isolation. The breakdown reports which transactions may have experienced anomalies and which transactions definitely did not. COCO does this in a lightweight fashion that introduces minimal overhead, allowing developers to monitor their production systems and tune the isolation levels on the fly. According to the LT-OCC protocol, a transaction having TID_{SI} being equal to the normal TID is serializable because all accesses occur at the same logical time. In the transaction validation phase, COCO applies this lightweight equality check to all snapshot transactions to detect transactions that may have observed concurrency anomalies (i.e., the ones with two different timestamps).

4.5 Implementation

This section describes COCO's underlying data structures, disk logging and checkpointing for durability and recovery, and implementation of insert and delete database operations.

4.5.1 Data Structures

COCO is a distributed in-memory OLTP database, in which each table in COCO has a pre-defined schema with typed and named attributes. Transactions are submitted to the system through pre-compiled *stored procedures* with different parameters, as in many popular systems [58, 97, 98, 105, 108]. Arbitrary logic (e.g., read/write and insert/delete operations) can be implemented in a stored procedure in C++.

Tables are currently implemented as a collection of hash tables — a primary hash

table and zero or more secondary hash tables. A record is accessed through the probing primary hash table. Two probes are needed for secondary index lookups, i.e., one in a secondary hash table to find the primary key, followed by a lookup on the primary hash table. The system currently does not support range queries, but can be easily adapted to tree structures [11, 62, 103]. Note that the two concurrency control algorithms [98, 108] have native support for range queries with phantom prevention.

In COCO, we use TIDs to detect conflicts. We record the TID value as one or more 64-bit integers depending on the underlying concurrency control algorithm, and the TID is attached to each record in a table's primary hash table. We show the TID format in PT-OCC in Figure 4-6. The high bits of each TID contain an epoch number, which indicates the epoch that a transaction comes from. The middle bits are used to distinguish transactions within the same epoch. The remaining two bits are the status bits showing if the record has been deleted or locked. Likewise, we use two 64-bit integers as a TID to represent `wts` and `rts` in LT-OCC. Note that status bits only exist in the integer indicating the `wts`. By default, the epoch size is 10 ms. The number of bits reserved for epochs is sufficient for ten years, and the number of bits reserved for transactions are sufficient for over 100 million transactions per epoch.

4.5.2 Disk Logging and Checkpointing

As shown in Section 4.2.2, a commit record is written to disk when an epoch commits. On recovery, the system uses the commit record to decide the outcome of the whole epoch. However, the commit record does not have the writes of each transaction from an epoch. As a result, COCO requires that the underlying concurrency control algorithm must properly log to disk as well.

We now take PT-OCC as an example to show how transactions are logged to disk. In COCO, each transaction is run by a single worker thread, which has a local recovery log. The writes of committed transactions are first buffered in memory. They are flushed to disk when the local buffer fills or when the system enters to the next epoch. A log entry contains the information of a single write to a record in

the database with the following information: (1) table and partition IDs, (2) TID, (3) primary key, and (4) value. A log entry in LT-OCC is the same as in PT-OCC, except for the TID, which has both `wts` and `rts`.

To bound the recovery time, a separate checkpointing thread can be used to periodically checkpoint the database to disk as in SiloR [113]. The checkpointing thread first logs the current epoch number e_c to disk, and next scans the whole database and logs each record to disk. Note that the epoch number e_c indicates when the checkpoint begins, and all log entries with embedded epoch numbers smaller than e_c can be safely deleted after the checkpoint finishes.

On recovery, COCO recovers the database through checkpoints and log entries. The system first loads the most recent checkpoint if available, and next replays all epochs since the checkpoint. An epoch is only replayed if the commit record indicates committed. Note that the database can be recovered in parallel with multiple worker threads. A write can be applied to the database as long as its TID is larger than the latest one in the database.

4.5.3 Deletes and Inserts

A record deleted by a transaction is not immediately deleted from the hash table. Instead, the system only marks the record as deleted by setting the delete status bit and registers it for garbage collection. This is because other concurrent transactions may read the record for validation. If a transaction finds a record it read has been deleted, it aborts and retries. All marked records from an epoch can be safely deleted, when the system enters the next epoch.

When a transaction makes an insert during execution, a placeholder is created in the hash table with TID 0. This is because a transaction needs to lock each record in its write set. If a transaction aborts due to conflicts, the placeholder is marked as deleted for garbage collection. Otherwise, the transaction writes a new value to the placeholder, updates the TID, and unlocks the record.

4.6 Evaluation

In this section, we study the performance of each distributed concurrency control algorithm with 2PC and epoch-based commit focusing on the following key questions:

- How does each distributed concurrency control algorithm perform with 2PC and epoch-based commit?
- What's the effect of durable write latency affect on 2PC and epoch-based commit?
- How does network latency affect each distributed concurrency control algorithm?
- How much performance gain of snapshot isolation over serializability?
- What's the overhead of different epoch sizes?
- How effective is each optimization technique?

4.6.1 Experimental Setup

We run our experiments on a cluster of eight `m5.4xlarge` nodes on Amazon EC2 [2], each with 16 2.50 GHz virtual CPUs and 64 GB RAM. Each node runs 64-bit Ubuntu 18.04 with Linux kernel 4.15.0. `iperf` shows that the network between each node delivers about 4.8 Gbits/s throughput. We implement COCO in C++ and compile it using GCC 7.4.0 with `-O2` option enabled.

In our experiments, we run 12 worker threads and 2 threads for network communication on each node. Each worker thread has an integrated workload generator. Aborted transaction are re-executed with an exponential back-off strategy. All results are the average of five runs.

Workloads

To evaluate the performance of COCO, we run a number of experiments using the following two popular benchmarks:

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) is a simple transactional workload. It's designed to be a benchmark for facilitating performance comparisons

of database and key-value systems [18]. There is a single table and each row has ten attributes. The primary key of the table is a 64-bit integer and each attribute has 10 random bytes. We run a workload mix of 80/20, i.e., each transaction[1] has 8 read operations and 2 read/write operation. By default, we run this workload with 20% multi-partition transactions that access to multiple partitions.

TPC-C: The TPC-C benchmark is a popular benchmark to evaluate OLTP databases [1]. It models a warehouse-centric order processing application. We support the `NewOrder` and the `Payment` transaction in this benchmark, which involves customers placing orders and making payments in their districts within a local warehouse. 88% of the standard TPC-C mix consists of these two transactions. We currently do not support the other three transactions that require range scans. By default, a NewOrder transaction is followed by a Payment transaction, and 10% of NewOrder and 15% of Payment transactions are multi-partition transactions.

In YCSB, we set the number of records to 400K per partition and the number of partitions to 96, which equals to the total number of worker threads in the cluster. In TPC-C, we partition the database by warehouse and there are 96 warehouses in total. We set the number of replicas to 3 in the replicated setting, i.e., each partition has a primary partition and two backup partitions, which are always hashed to three different nodes.

Distributed concurrency control algorithms

We study the following distributed concurrency control algorithms in COCO. To avoid an apples-to-oranges comparison, we implemented all algorithms in C++ in our framework.

S2PL: This is a distributed concurrency control algorithm based on strict two-phase locking. Read locks and write locks are acquired as a worker runs a transaction. To avoid deadlock, the same `NO_WAIT` policy is adopted as discussed in Section 4.3. A worker thread updates all records and replicates the writes to replicas before releasing

[1] YCSB+T [27, 28], another extension to YCSB, wraps operations within transactions in a similar way to model activities happened in a closed economy.

Table 4.1: Concurrency control algorithms, commit protocols and replication schemes supported in COCO

	2PC w/o Rep.	2PC w/ Sync.	Epoch w/ Async.
S2PL	✓	✓	
PT-OCC	✓	✓	✓
LT-OCC	✓	✓	✓

all acquired locks.

PT-OCC: Our physical time OCC from Section 4.4.1.

LT-OCC: Our logical time OCC from Section 4.4.2.

In addition to each concurrency control algorithm, we also support three different combinations of commit protocols and replication schemes.

2PC w/o Replication: A transaction commits with two-phase commit and no replication exists.

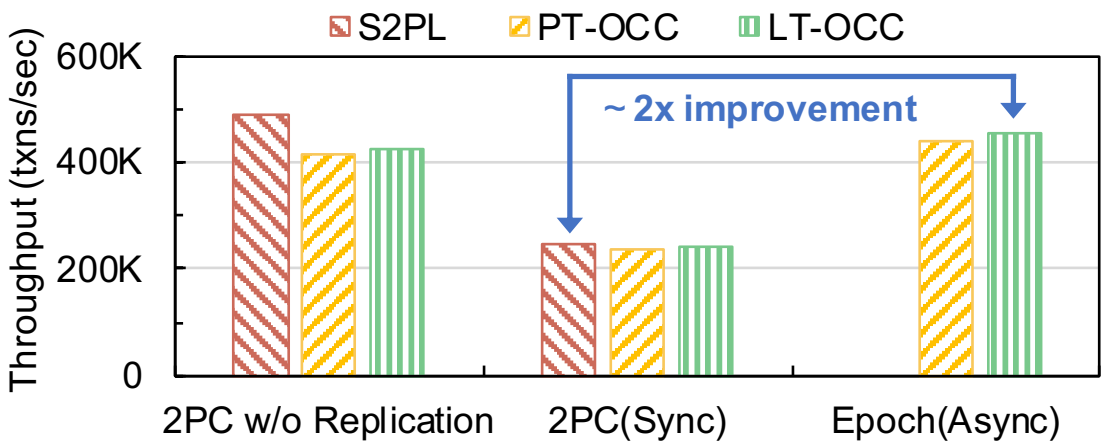
2PC(Sync): A transaction commits with two-phase commit. A transaction's write locks are not released until all writes are replicated on all replicas.

Epoch(Async): A transaction commits with epoch-based commit and replication, i.e., a transaction's write locks are released as soon as the writes are applied to the primary replica.

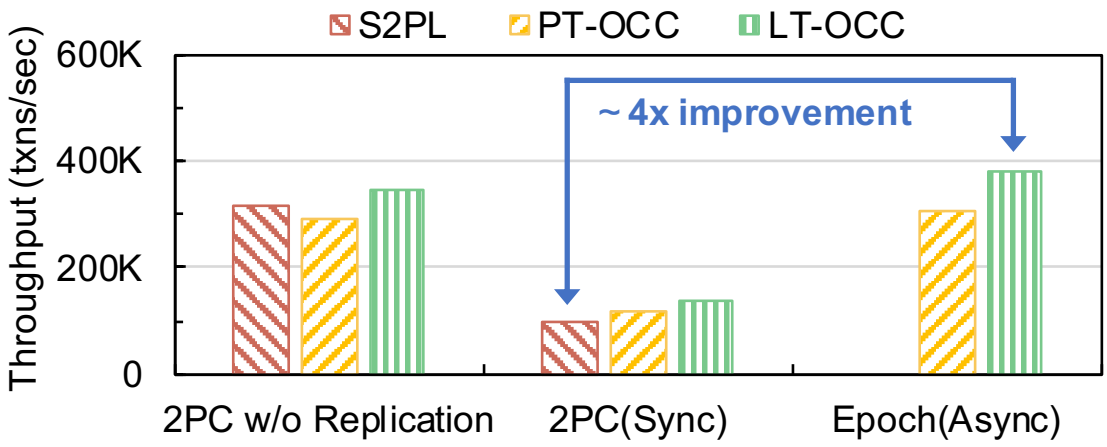
We summarize the supported concurrency control algorithms, commit protocols and replication schemes in Table 4.1. Note that a transaction cannot run with S2PL and commit with *Epoch(Async)*, since it's not straightforward to asynchronously apply writes on replicas without write locks being held in S2PL. By default, optimistic concurrency control protocols (i.e., PT-OCC and LT-OCC) are allowed to read from the nearest replica in both *2PC(Sync)* and *Epoch(Async)*.

4.6.2 Performance Comparison

We now study the performance of each concurrency control algorithm with different combinations of commit protocols and replication schemes.



(a) YCSB



(b) TPC-C

Figure 4-7: Performance of 2PC w/o Replication, 2PC(Sync) and Epoch(Async)

We first ran a YCSB workload and report the result in Figure 4-7(a). S2PL with synchronous replication only achieves about 50% of the original throughput when there is no replication. This is because each transaction cannot commit before writes are replicated and even a single-partition transaction now has a round-trip delay. Similarly, both PT-OCC and LT-OCC have about 40% performance slowdown, even though they are allowed to read from the nearest replica. We now study how epoch-based commit and replication affect the performance of PT-OCC and LT-OCC, which is shown in *Epoch(Async)*. As shown in the right side of Figure 4-7(a), both PT-OCC and LT-OCC have about 2x performance improvement compared to the ones (shown

in the middle) with *2PC(Sync)*. This is because the write locks can be released as soon as the writes have been applied at the primary replica. In this way, transactions no longer pay the cost of a round-trip delay.

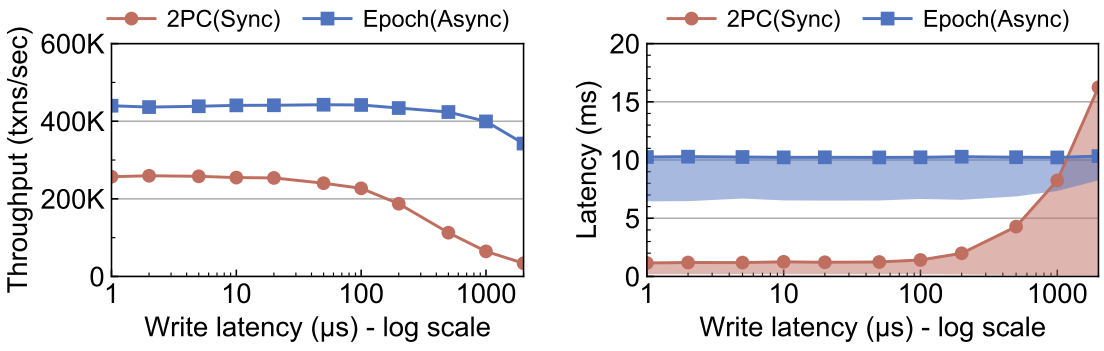
We also ran a TPC-C workload and report the result in Figure 4-7(b). Similarly, we observe that there exist a 4x performance improvement from *Epoch(Async)* over *2PC(Sync)*. The throughput even exceeds the ones (shown in the left side) with *2PC w/o Replication*, since PT-OCC and LT-OCC are allowed to read from the nearest replica.

In summary, concurrency control protocols are able to achieve 2 $\sim$ 4x higher throughput through epoch-based commit and replication compared to the ones with *2PC(Sync)*.

4.6.3 Effect of Durable Write Latency

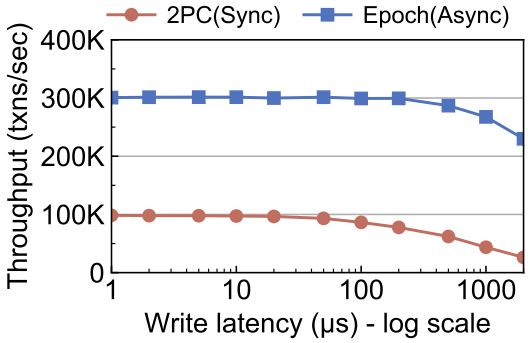
We next study the effect of durable write latency on throughput and latency. For databases with high availability, the prepare and commit records of 2PC and epoch-based commit must be written to durable secondary storage, such as disks and replication to a remote node. To model the latency of various durable secondary storage, we add an artificial delay to 2PC and epoch-based commit through spinning.

In this experiment, we vary the durable write latency from 1 μs to 2 ms and run both YCSB and TPC-C. Figure 4-8(a) and 4-8(b) show the throughput and the latency at the 99th percentile of *2PC(Sync)* and *Epoch(Async)* on YCSB. For interested readers, we also report the latency at the 50th percentile, which is shown at the bottom of the shaded band. Transactions with *2PC(Sync)* have a noticeable throughput decline or a latency increase when the durable write latency is larger than 20 μs. In contrast, transactions with *Epoch(Async)* have a stable throughput until the durable write latency exceeds 200 μs. They also always have a stable latency, since the epoch size can be dynamically adjusted based on different durable write latency. Likewise, we report the result of TPC-C in Figure 4-8(c) and 4-8(d). A noticeable throughput decline or a latency increase is observed on *2PC(Sync)* when the durable write latency is larger than 50 μs.

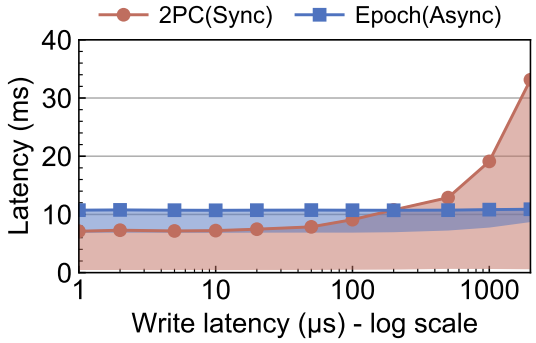


(a) YCSB - Throughput

(b) YCSB - Latency



(c) TPC-C- Throughput



(d) TPC-C- Latency

Figure 4-8: Performance of 2PC(Sync) and Epoch(Async) with varying durable write latency

Overall, epoch-based commit and replication trade latency for higher throughput. The performance advantage is more significant when durable write latency is larger. For example, with 1 ms durable write latency, *Epoch(Async)* has roughly 6x higher throughput than *2PC(Sync)* on both YCSB and TPC-C.

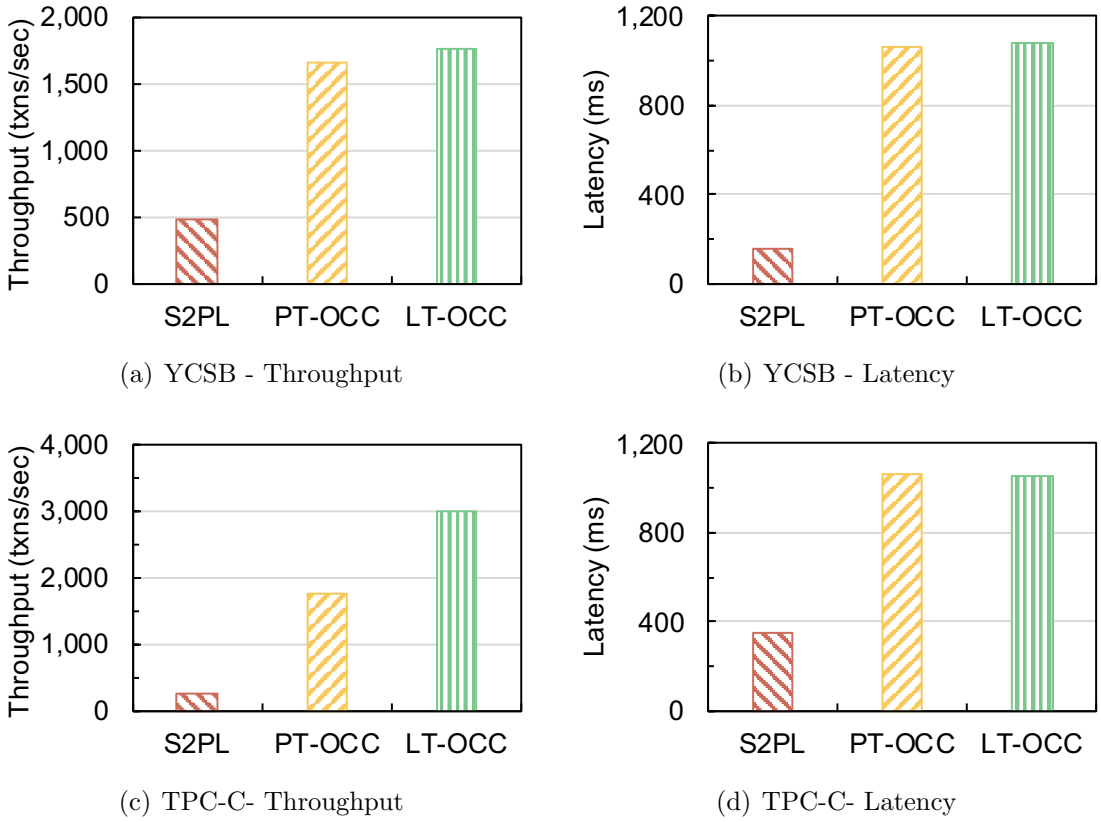
4.6.4 Wide-Area Network Experiment

In this section, we study how epoch-based commit and replication perform compared to S2PL with *2PC(Sync)* in the wide-area network (WAN) setting. For users concerned with very high availability, wide-area replication is important because it allows the database to survive the failure of a whole data center (e.g., due to a power outage or a natural disaster).

We use three m5.4xlarge nodes running on Amazon EC2 [2]. The three nodes

Table 4.2: Round trip times between EC2 nodes

	Latency (ms)		
	N. Virginia	Ohio	N. California
N. Virginia	-	11.326	60.949
Ohio	11.314	-	50.002
N. California	60.957	50.043	-



(a) YCSB - Throughput

(b) YCSB - Latency

(c) TPC-C- Throughput

(d) TPC-C- Latency

Figure 4-9: Performance of 2PC(Sync) and Epoch(Async) in a wide-area network

are in North Virginia, Ohio, and North California respectively. Each partition of the database is fully replicated in all area zones, meaning each one has a primary partition and two backup partitions. The primary partition is randomly chosen from 3 nodes. The round trip times between any two nodes are shown in Table 4.2. In this experiment, we set the epoch size to one second, and use *2PC(Sync)* in S2PL and *Epoch(Async)* in PT-OCC and LT-OCC.

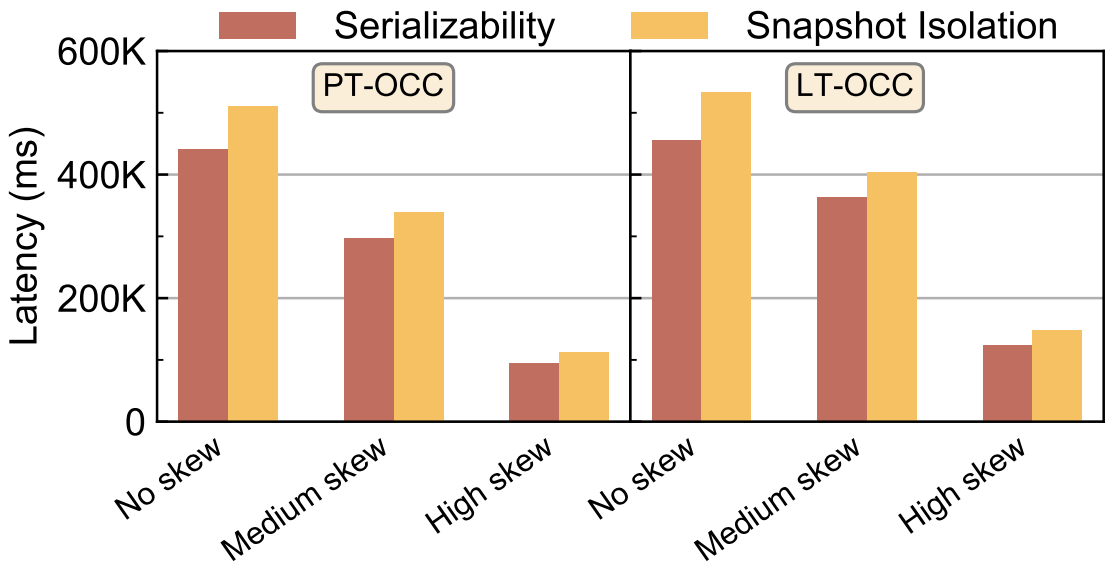


Figure 4-10: Snapshot transactions

In this experiment, we ran both YCSB and TPC-C. We report the throughput and the latency at the 99th percentile in Figure 4-9. As shown in Figure 4-9(a) and Figure 4-9(c), the throughput of *Epoch(Async)* is 4x higher than S2PL with *2PC(Sync)* on YCSB. On TPC-C, the throughput improvement is up to one order of magnitude. In contrast, S2PL has lower latency at the 99th percentile than PT-OCC and LT-OCC. For example, transactions with *2PC(Sync)* only have 150 ms latency on YCSB and 350 ms latency on TPC-C as shown in Figure 4-9(b) and Figure 4-9(d). The latency at the 99th percentile in *Epoch(Async)* is about one second due to the nature of epoch-based commit and replication.

In summary, the performance advantage of epoch-based commit and replication over *2PC(Sync)* is more significant in the WAN setting.

4.6.5 Snapshot Transactions

We now study how much performance gain that PT-OCC and LT-OCC are able to achieve when they run under snapshot isolation versus serializability. In this experiment, we report the result on YCSB in Figure 4-10. To increase read/write conflicts, we make each access follow a Zipfian distribution [41].

We consider three different skew factors: (1) No skew (0), (1) Medium skew (0.8),

and (3) High skew (0.999). As there is more contention with a larger skew factor, the performance of both algorithms under both snapshot isolation and serializability goes down. This is because the system has a higher abort rate when the workload becomes more contended. Meanwhile, both PT-OCC and LT-OCC have higher throughput under snapshot isolation because of a lower abort rate and the parallel locking and validation optimization as discussed in Section 4.4. Overall, both PT-OCC and LT-OCC have about 20% higher throughput running under snapshot isolation than serializability.

4.6.6 Effect of Epoch Size

We next study the effect of epoch size on both throughput and latency. We varied the epoch size from 1 ms to 100 ms, and report the throughput and the latency at the 99th percentile of PT-OCC on YCSB in Figure 4-11. The result of LT-OCC is almost the same and is not reported.

When an epoch-based commit happens, the system uses a global barrier to guarantee that all writes of committed transactions are replicated across all replicas. Intuitively, the system spends less time on synchronization with a larger epoch size. As we increase the epoch size, the throughput of PT-OCC continues increasing and levels off beyond 50 ms. The latency at the 99th percentile roughly equals to the epoch size.

In summary, with an epoch size of 10 ms, the system can achieve more than 93% of its maximum throughput (the one achieved with a 100 ms epoch size) and have a good balance between throughput and latency.

4.6.7 Factor Analysis

At last, we study the effectiveness of each optimization technique in more details through a factor analysis. We run the YCSB workload and report the result in Figure 4-12.

We introduce one technique at a time to *2PC(Sync)*, in which PT-OCC and

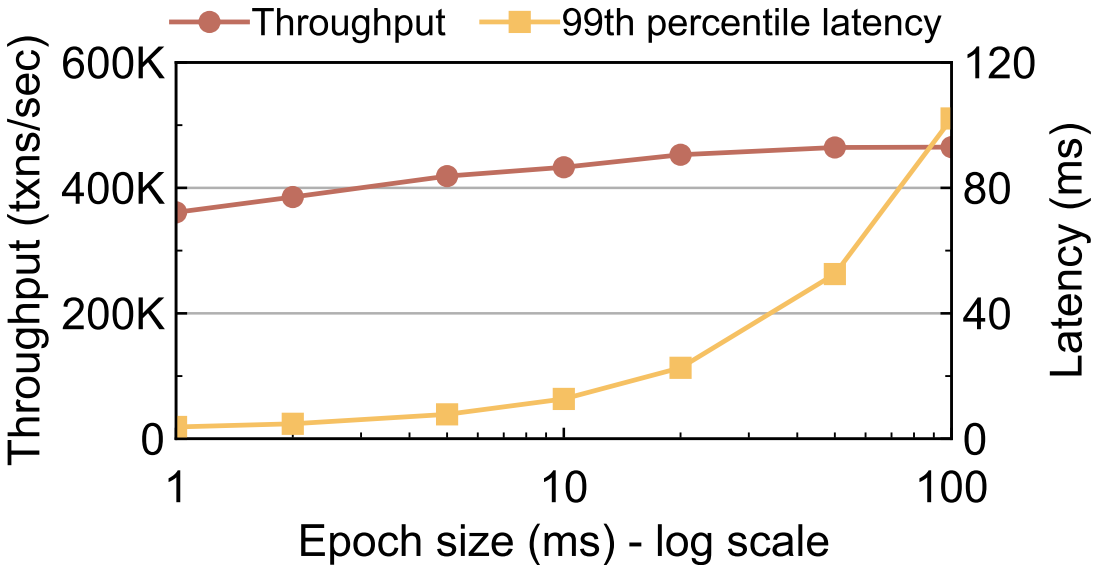


Figure 4-11: Effect of epoch size

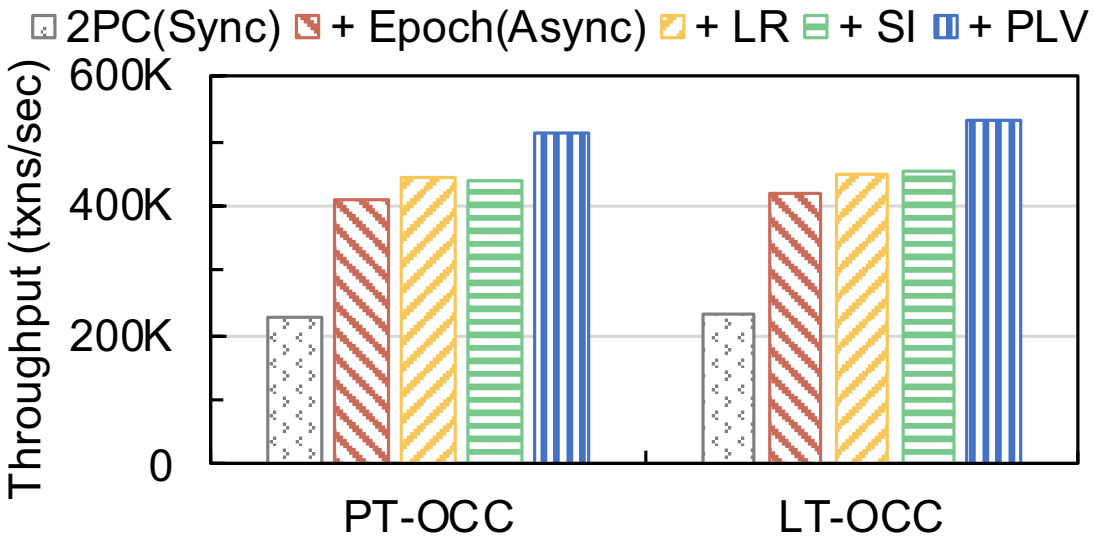


Figure 4-12: Factor analysis

LT-OCC use 2PC and synchronous replication. *+ Epoch(Async)* refers to using epoch-based commit and replication. However, we do not allow transactions to read from the nearest replica in this setting. *+ LR* refers to the technique that allows transactions to read from the nearest replica. *+ SI* refers to running transactions under snapshot isolation. *+ PLV* refers to the technique that allows locking and read validation in parallel for snapshot transactions.

From all optimization techniques above, *+ Epoch(Async)* has the most significant performance improvement – about 80% over *2PC(Sync)*. *+ SI* has little impact on the throughput, since the workload is not contended and running transactions under snapshot isolation does not further improve the performance. However, *+ PVL* is still able to achieve about 16% higher throughput due to reduced round-trip delays.

Overall, PT-OCC and LT-OCC with all optimization techniques achieve up to 2x higher throughput than the ones with *2PC(Sync)*.

4.7 Related Work

The design of COCO is inspired by many pieces of related work, including transaction processing, consistency and replication.

4.7.1 Transaction Processing

The seminal survey by Bernstein et al. [7] summarizes classic distributed concurrency control protocols. As in-memory databases become more popular, there has been a resurgent interest in transaction processing in both multicore processors [47, 54, 98, 106, 108] and distributed systems [20, 42, 59, 67, 87]. Many recent transactional databases [21, 58, 98] employ epochs to trade latency for higher throughput. Silo [98] reduces shared-memory writes through not detecting anti-dependencies within epochs. Obladi [21] delays updates within epochs to increase system throughput and reduce bandwidth cost. STAR [58] separates the execution of single-node and distributed transactions and runs them in different epochs. COCO is the first to leverage epochs to reduce the cost of two-phase commit (2PC) for distributed transactions and achieves strong consistency between replicas with asynchronous writes. Warranties [56] reduces coordination on read validation by maintaining time-based leases to popular records, but writes have to be delayed. In contrast, LT-OCC in COCO reduces coordination without penalizing writes. This is because writes instantly make the read validity timestamps on old records expired.

4.7.2 Replicated Systems

High availability is typically implemented through replication. Paxos [50] and other leader-less consensus protocols such as EPaxos [66] are popular solutions to coordinate the concurrent reads and writes to different copies of data while providing consistency. Spanner [19] is a Paxos-based transaction processing system based on a two-phase locking protocol. Each master replica initiates a Paxos protocol to synchronize with backup replicas. The protocol incurs multiple round-trip messages for data accesses and replication coordination. TAPIR [112] eliminates the overhead of Paxos by allowing inconsistency in the storage system and building consistent transactions using inconsistent replication. Ganymed [75, 76] runs update transactions on a single node and propagates writes of committed transactions to a potentially unlimited number of read-only replicas. In COCO, the writes of committed transactions are asynchronously applied to each replica and transactions are allowed to read from the nearest replica to reduce round-trip communication. Some recent systems [48, 60, 71, 107] optimize the latency and number of round trips needed to commit transactions across replicas. MDCC [48] is an OCC protocol that exploits generalized Paxos [51] to reduce the coordination overhead, in which a transaction can commit with a single network round trip in the normal operation. Replicated Commit [60] needs less round-trip communication by replicating the commit operation rather than the transactional log. In contrast, COCO avoids the use of 2PC and treats an epoch of transactions as the commit unit that amortizes the cost of 2PC.

4.7.3 Consistency and Snapshot Isolation

Due to the overhead of implementing strong isolation, many systems use weaker isolation levels instead (e.g., PSI [90], causal consistency [64], eventual consistency [94], or no consistency [79]). Lower isolation levels trade programmability for performance and scalability. In this chapter, we focus on serializability and snapshot isolation, which are the gold standard for transactional applications. By maintaining multiple data versions, TxCache [77] ensures that a transaction always reads from a consistent

snapshot regardless of whether each read operation comes from the database or the cache. Binning et al. [12] show how only distributed snapshot transactions pay the cost of coordination. Serial Safety Net (SSN) [100] is able to make any concurrency control protocol to support serializability by detecting dependency cycles. RushMon [88] reports the number of isolation anomalies based on the number of cycles in the dependency graph, using a sampling approach. Similar, LT-OCC in COCO is also able to detect concurrency anomalies when running snapshot transactions through a simple equality check.

4.8 Summary

In this chapter, we presented COCO, a new distributed OLTP database, which enforces atomicity, durability and consistency at the boundaries of epochs. By separating transactions into epochs and treating a whole epoch of transactions as the commit unit, COCO is able to address inefficiency found in conventional distributed databases with two-phase commit and synchronous replication. In addition, the system supports two variants of optimistic concurrency control (OCC) using physical time and logical time with various optimizations, which are enabled by epoch-based commit and replication. Our results on two popular benchmarks show that COCO outperforms systems with conventional commit and replication schemes by up to a factor of four.

Chapter 5

Aria: A Fast and Practical Deterministic OLTP Database

5.1 Introduction

Modern database systems employ replication for high availability and data partitioning for scale-out. Replication allows systems to provide high availability, i.e., tolerance to machine failures, but also incurs additional network round trips to ensure writes are synchronized to replicas. Partitioning across several nodes allows systems to scale to larger databases. However, most implementations require the use of two-phase commit (2PC) [65] to address the issues caused by nondeterministic events such as system failures and race conditions in concurrency control. This introduces additional latency to distributed transactions and impairs scalability and availability (e.g., due to coordinator failures).

Deterministic concurrency control algorithms [35, 36, 96, 97] provide a new way of building distributed and highly available database systems. They avoid the use of expensive commit and replication protocols by ensuring different replicas always *independently* produce the same results as long as the same input transactions are given. Therefore, rather than replicating and synchronizing the updates of distributed transactions, deterministic databases only have to replicate the input transactions across different replicas, which can be done asynchronously and often with much less com-

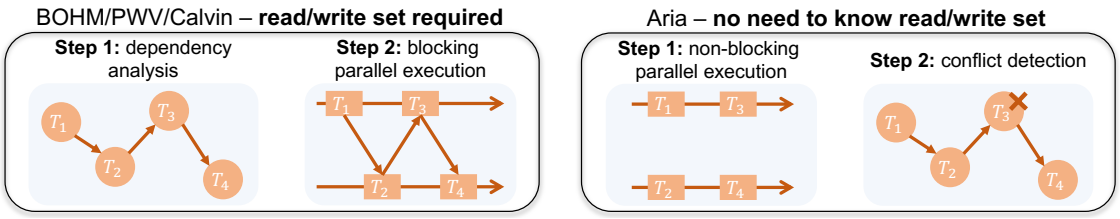


Figure 5-1: Comparison of Aria, BOHM, PWV, and Calvin

munication. In addition, deterministic databases avoid the use of two-phase commit, since they naturally eliminate nondeterministic race conditions in concurrency control and are able to recover from system failures by re-executing the same original input transactions.

The state-of-the-art deterministic databases — BOHM [36], PWV [35], and Calvin [97] — achieve determinism through *dependency graphs* or *ordered locks*. The key idea in BOHM and PWV is that a dependency graph is built from a batch of input transactions based on the read/write sets. In this way, the database can produce deterministic results as long as the transactions are run following the dependency graph. The key idea in Calvin is that read/write locks are acquired prior to executing the transaction, and according to the ordering of input transactions. A transaction is assigned to a worker thread for execution once all needed locks are granted. As shown in the left side of Figure 5-1, existing deterministic databases perform dependency analysis before transaction execution, which requires that the read/write set of a transaction be known a priori. For very simple transactions, e.g., that only access to records via equality lookups on a primary key, this can be done easily. However, in reality, many transactions access records through complex predicates over non-key attributes; for such queries, these systems must execute the query at least twice: once to determine the read/write set, once to execute the query, and possibly more times if the pre-determined read/write set changes between these two executions. In addition, Calvin requires the use of a single-threaded lock manager per database partition, which significantly limits the concurrency it can achieve.

In this chapter, we propose a new system, Aria, to address the limitations in previous deterministic OLTP databases with a fundamentally different mechanism,

which does not require any analysis or pre-execution of input transactions. Aria runs transactions in batches. The key idea is that each replica runs an identical batch of transactions on an identical database snapshot, and resolves conflicts in the same way, ensuring deterministic execution. As shown in the right side of Figure 5-1, each replica reads from the current snapshot of the database and executes all transactions to completion in an *execution phase*, and then chooses deterministically which transactions should commit and which should abort to ensure serializability in a *commit phase*. For good scalability and high transaction throughput, the serializability check in the commit phase is performed in parallel on each transaction independently and no central coordination is required. Aborted transactions will be scheduled for re-execution at the beginning of the next batch. In this way, Aria can enforce determinism without needing to know the read or write sets of input transactions in advance. Note that although optimistic concurrency control (OCC) algorithms [49] also resolve conflicts after execution, transactions in OCC can commit in a non-deterministic order depending on scheduling variations, which is fundamentally different from Aria.

We also introduce a novel deterministic reordering mechanism that brings a new opportunity for reducing conflicts and improving the throughput of Aria. Unlike existing deterministic databases [36, 96] that execute transactions strictly following the ordering of the input, our reordering mechanism allows Aria to commit transactions in an order that reduces aborts. Furthermore, the reordering is done in parallel as a part of the process of identifying aborted transactions and imposes minimal execution overhead.

Our evaluation on a single multicore node shows that Aria outperforms conventional nondeterministic concurrency control algorithms and state-of-the-art deterministic databases by a large margin on YCSB [18]. Further, Aria is able to achieve higher transaction throughput by up to a factor of three with deterministic reordering on a YCSB workload with skewed access. On a subset of TPC-C [1], in which more conflicts exist, Aria achieves competitive performance to PWV and higher performance than all other deterministic databases. Our evaluation on a cluster of eight nodes running on Amazon EC2 [2] shows that Aria outperforms Calvin by up to a factor of

two on YCSB and TPC-C. Finally, we show Aria achieves near linear scalability to 16 nodes.

5.2 State-of-the-art Deterministic Databases

One way to achieve determinism is to allow only one thread to run transactions [69]. However, support for multiple threads in modern database systems is crucial, as they allow hiding of I/O latency [8] and exploiting of multi-core parallelism [98]. More recently, deterministic protocols based on dependency graphs [35, 36] or ordered locks [96, 97] have been proposed to provide more parallelism while ensuring determinism. Although the systems above take an important step towards building deterministic databases, they still have one limitation that makes them impractical: the dependency analysis requires that the read/write sets of a transaction (i.e., a collection of primary keys) must be known a priori before execution. For transactions with secondary index lookups or other data dependencies, it may not be possible to determine reads/writes in advance. In such cases, a transaction must run over the database to determine its read/write sets. The transaction then aborts and uses these pre-computed read/write sets for re-execution. However, the transaction may be executed multiple times if any key in the read/write sets changes during this re-execution. We next discuss why the design of existing deterministic databases can lead to undesirable performance.

BOHM, a single-node deterministic database, runs transactions in two steps. In the first step, it inserts placeholders for the primary keys in each input transaction's write set along with the transaction ID (TID) into a multi-version storage layer. In the second step, a transaction must read a particular version for each key in its read set — the one with the largest TID up to the transaction's TID — to enforce determinism. When all the keys in a transaction's read set are ready, the transaction can execute and update the placeholders that were inserted earlier. To avoid contention when inserting placeholders into the storage layer, each worker thread in BOHM scans all input transactions to look for the primary keys for update in its own partition.

PWV, a single-node deterministic database, first decomposes each input transaction into a set of pieces (i.e., sub-transactions) such that each piece only reads from or writes into one partition. It next builds a dependency graph, in which an edge indicates conflicts between pieces within a partition, and uses it to schedule the execution of each piece to meet the requirement of both data dependencies and commit dependencies. Different from BOHM, PWV commits the writes of each transaction at a finer granularity of pieces instead of the whole transaction. In this way, later transactions spend less time waiting to see the writes from earlier transactions at the cost of a more expensive fine-grained dependency graph analysis.

Calvin, a distributed deterministic database, uses a single-threaded lock manager to scan the input transactions and grants read/write locks based on pre-declared read/write sets. Once all locks of a transaction are acquired, the lock manager next assigns the transaction to an available worker thread for execution. Once the execution finishes, the worker thread releases the locks. The separate roles of lock manager threads and worker threads increase the system's overall synchronization cost, and locks on one partition must be granted by a single lock manager thread. In machines with many cores, it is difficult to saturate system resource with one single-threaded lock manger, as many worker threads are idle waiting for locks. To solve this issue, the database can be partitioned into multiple partitions per machine, but this introduces additional overhead by introducing more multi-partition transactions with overhead due to redundant execution [97]. For example, a transaction updating both y_1 and y_2 to $f(x)$ must run $f(x)$ twice, if locks on y_1 and y_2 are granted by two different lock mangers.

Note that both BOHM and PWV enforce determinism through dependency graphs, which are built before transaction execution. Calvin runs transactions following dependency graphs as well, but it is achieved implicitly through ordered locks. We will discuss how Aria achieves determinism and addresses the issues above in the following sections.

```
1  Data: write reservations: writes
2
3  ### the execution phase of batch i ###
4  for each transaction T in batch i:
5     Execute(T, db)
6     ReserveWrite(T, writes)
7
8  ### the commit phase of batch i ###
9  for each transaction T in batch i:
10    Commit(T, db, writes)
```

Figure 5-2: Execution and commit phases in Aria

5.3 System Overview

We now give a high level overview how Aria achieves deterministic execution. As in existing deterministic databases, replicas in Aria do not need to communicate with one another, and run transactions independently. This is because the same results will always be produced under deterministic execution. Each transaction passes through a sequencing layer and is assigned a unique transaction ID (TID). The TID indicates a total ordering among transactions; by default it indicates the commit order of transactions, but our deterministic reordering technique, which we will describe in Section 5.5, may commit transactions in a different order.

Aria runs transactions in batches in two different phases: an *execution phase* and a *commit phase*, separated by a barrier. Within a batch, each Aria replica runs transactions in parallel and in any order with multiple worker threads. The transactions are assigned to worker threads in a round-robin fashion. As shown in Figure 5-2, in the execution phase, each transaction reads from the current snapshot of the database and keeps the writes in a local write set. Unlike BOHM, Aria adopts a single-version approach, even though it buffers writes for transactions until the end of a batch. Once all transactions in the batch finish execution on a replica, it enters the commit phase. The commit phase on a replica also runs on multiple worker threads, each executing independently. In the commit phase, a thread aborts transactions that per-

formed conflicting operations with *earlier* transactions, i.e., those with *smaller* TIDs. For example, a transaction would abort if it reads a record modified by some earlier transaction. Aborted transactions are automatically scheduled at the beginning of the next batch for execution, unless the transaction is aborted explicitly (e.g., for violating some integrity constraint). If a transaction does not have conflicts with earlier transactions, the system applies its changes to the database. Consider the example in Figure 5-3, with four transactions in a batch. Transaction T_2 has conflicts with T_1. To ensure serializability, transaction T_2 aborts and is scheduled at the beginning of the next batch for execution. Note that i) each replica has identical state at the start of each batch; ii) each replica commits transactions in the same order (no matter which order workers run transactions in), and, iii) transactions within a batch never see each other's writes. Therefore, transactions across replicas will always read and write the same values within a batch. In addition, each replica chooses to commit and abort the same transactions. Thus Aria is deterministic.

In addition to resolving conflicts following the ordering of input transactions, Aria uses a deterministic reordering technique that reduces conflicts in the commit phase. For example, consider a sequence of n transactions, in which T_i reads record i and updates record $i + 1$, for $0 < i \leq n$. Thus, each transaction (except T_1) reads a record that has been modified by the previous transaction. Following the default algorithm in Aria, the first transaction is the only transaction that can commit. However, these "conflicting" transactions can still commit under the serial order $T_n \rightarrow T_{n-1} \rightarrow \cdots \rightarrow T_1$. With deterministic reordering, Aria does not physically reorder the input transactions. Instead, it commits more transactions so that the results are still serializable but equivalent to a different ordering.

5.4 Deterministic Batch Execution

In this section, we describe the details of how Aria runs transactions deterministically in two phases, such that the serial order of committed transactions strictly follows the ordering of input transactions. In Section 5.5, we will present the reordering

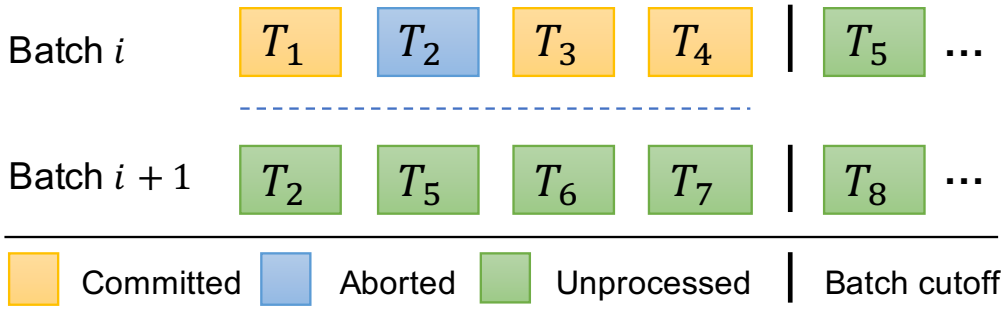


Figure 5-3: Batch processing in Aria

technique to relax this constraint. We next give a proof to show that Aria produces serializable results. We also show how transactions can run under snapshot isolation with minor changes to the protocol. Finally, we discuss the phantom problem and limitations.

5.4.1 The Execution Phase

During this phase, each transaction reads from the current snapshot of the database, executes its logic, and writes the result to a local write set, as shown from line 1 to line 3 in Figure 5-4. Since the changes made by transactions are kept locally and are not directly committed to the database, the snapshot read by each transaction in the execution phase is always the same.

Once a transaction finishes execution, it goes through its local write set and makes *write reservations* for each entry in its write set, as shown from line 5 to line 7 in Figure 5-4.

A write reservation on a previously reserved value can be made only when the reserving transaction has a smaller TID than the previous reserver. If a reservation from a larger TID already exists, the old reservation will be voided and a new reservation with the smaller TID will be made. If a transaction cannot make a reservation, the transaction must abort. In addition, the transaction can also skip the commit phase as a performance optimization, since it knows at least one reservation is not successful. Note that a transaction is not allowed to skip the rest of reservations due to an earlier unsuccessful reservation and must continue making all reservations. This

is because reservations are made in parallel by multiple worker threads and omitting the rest of reservations will produce nondeterministic results.

We now use an example to show how transactions make write reservations.

Example 1. *Consider the following three transactions:* T_1*:* $x = x+1$*,* T_2*:* $y = x-y$*, and* T_3*:* $x = x+y$

The reservation table is initially empty. Transaction T_1 first makes reservations for its writes. The reservation succeeds and the table now has an entry $\{x : T_1\}$. Transaction T_2 then makes its reservation successfully by creating an entry $\{y : T_2\}$ in the reservation table. Finally, transaction T_3 tries to make a reservation for record x. Since record x has been reserved by transaction T_1, and T_3 has a larger TID, the reservation fails and T_3 must abort. Note that, even though we describe the reservation process in a sequential manner, the whole process can be conducted in parallel and in any order, and the same result will always be produced.

Once all transactions finish execution and reservations are made for writes, the system enters the commit phase.

5.4.2 The Commit Phase

Aria separates concurrency control from transaction execution. During the commit phase, each transaction is determined to commit or abort deterministically based on the write reservations made in the execution phase. Aria does not require two-phase commit to commit distributed transactions as in non-deterministic databases. The reasons are twofold: (1) the write reservations made in the execution phase do not affect the value of each record in the database, i.e., no rollback is required, and (2) a transaction can always commit (i.e., by applying writes to the database) even a failure occurs, since determinism guarantees that the same result is always produced after re-execution.

We now introduce three types of dependencies that Aria uses to determine if a transaction can commit or not: (1) transaction T_i has a write-after-write (WAW) dependency on transaction T_j if T_i tries to update some record after T_j has updated

```
1  Function: Execute(T, db)
2        Read from the latest snapshot of db
3        Execute to compute T's read/write set # (RS & WS)
4
5  Function: ReserveWrite(T, writes)
6        for each key in T.WS:
7           writes[key] = min(writes[key], T.TID)
8
9  Function: Commit(T, db, writes)
10       # T aborts if writes are not installed
11       if HasConflicts(T.TID, T.RS ∪ T.WS, writes) == false:
12          Install(T, db)
13
14 Function: HasConflicts(TID, keys, reservations)
15       for each key in keys:
16          if key in reservations and reservations[key] < TID:
17             return true
18       return false
19
20 Function: Install(T, db)
21       for each key in T.WS:
22          write(key, db)
```

Figure 5-4: Execution and commit protocols in Aria

it, (2) transaction T_i has a write-after-read (WAR) dependency on transaction T_j if T_i tries to update some record after T_j has read it, and (3) transaction T_i has a read-after-write (RAW) dependency on transaction T_j if T_i tries to read some record after T_j has updated it.

Aria decides if a transaction T_i can commit or not by checking if it has certain types of dependencies with any earlier transaction T_j, $\forall j < i$. There are two types of dependencies that the system must check: (1) WAW-dependency: if T_i WAW-depends on some T_j, it must abort. This is because the updates are installed independently and more than one update goes to the same record, and (2) RAW-dependency: if T_i RAW-depends on some T_j, it must abort as well, because it should have seen T_j's write but did not. In contrast, it's safe for a transaction to update some record

that has been read by some earlier transaction. Thus, by default, Aria does not check WAR-dependencies, however, WAR-dependencies can help reduce dependency conflicts with deterministic reordering as will be noted in Section 5.5.

Note that by aborting transactions with WAW-dependencies or RAW-dependencies, Aria ensures that the serial order of committed transactions is the same as the input order.

Rule 1. *A transaction commits if it has no WAW-dependencies or RAW-dependencies on any earlier transaction.*

Aria does not have to go through all earlier transactions to check WAW-dependencies and RAW-dependencies. Instead, it uses a transaction's read set and write set to probe the write reservation table. Note that some transactions may have aborted in the execution phase and can simply skip the commit phase, e.g., a reservation on some write failed. As shown in Figure 5-4 (line 9 – 12), function `HasConflicts` returns false when none of the keys in a transaction's read set and write set exists in the write reservation table. When no dependencies exist, the transaction commits and all writes are applied to the database.

As in the execution phase, the process of checking dependencies can be done in parallel and in any order for every transaction. Aborted transactions are automatically scheduled at the beginning of the next batch for execution, unless the transaction is aborted explicitly (e.g., violating some integrity constraint). The relative ordering of aborted transactions is kept the same. As a result, every transaction can commit eventually, since the first transaction in a batch always commits and the position of an aborted transaction always moves forward across batches.

5.4.3 Determinism and Serializability

We now show how the system achieves determinism and serializability with batch execution.

Theorem 1. *Aria following Rule 1 enforces determinism and serializability.*

Determinism: In the execution phase, the system builds a write reservation table given a batch of transactions. The reservation table is a collection of key-value pairs. A key x is a record modified by some transaction and the value T is the transaction that modifies record x with the smallest TID. Specifically, no matter what order transactions execute in, the collection of key-value pairs always equals to

$$\big\{\{x \in \mathcal{WS} : T \in \mathcal{C}(x)\} \mid \mathcal{ID}(T) \leq \mathcal{ID}(T'), \forall T' \in \mathcal{C}(x)\big\}$$

where $\mathcal{WS}$ indicates the union of the write sets of all transactions and $\mathcal{C}(x)$ indicates a set of transactions that wrote record x. We use $\mathcal{ID}(T)$ to indicate transaction T's TID.

In the commit phase, the reservation table does not change and is used by the system to detect dependencies between transactions. Following the algorithm in Figure 5-4, whether a transaction will commit or abort depends only on its read/write sets and the reservation table regardless of the order in which transactions run in the commit phase. Since each replica runs an identical batch of transactions on an identical database snapshot, different replicas will produce the identical read/write sets and the reservation table. Therefore, Aria achieves determinism.

Serializability: The following proof is by contradiction.

Proof. (BY CONTRADICTION.) Assume the ordering of committed transactions is: $\cdots \to T_i \to \cdots \to T_j \to \ldots$, and the result produced by Aria is not serializable. Since transactions are committed by multiple worker threads in parallel, there are two possible outcomes: (1) transaction T_j's updates were overwritten by transaction T_i's updates, and (2) transaction T_j read transaction T_i's writes. Since WAW-dependencies do not exist, T_i's write set does not overlap with T_j's, so concurrent transactions must have updated different records. Similarly, since RAW-dependencies do not exist, the values of records that transaction T_j read must be the same as in the original database snapshot. By Rule 1, both outcomes lead to a contradiction. □

Thus, Aria without deterministic reordering commits transactions under conflict serializability, and the result of the database is equivalent to running all committed

transactions serially following the input order. Note that the properties enforced by Aria are sufficient but not necessary for serializability.

5.4.4 Transactions under Snapshot Isolation

Transactions running under snapshot isolation (SI) do not have to follow a serial order. Instead, a transaction is only required to run over some transactionally consist snapshot of the database. This means that a transaction can commit as long as it experiences no concurrent write-write conflicts, i.e., no WAW-dependencies exist.

In the snapshot isolation mode, Aria runs the same execution phase as in the serializable mode. In the commit phase, the system only checks WAW-dependencies and ignores all RAW-dependencies. Specifically, the second parameter of function `HasConflicts` at line 11 of Figure 5-4 is changed from `T.RS` $\bigcup$ `T.WS` to `T.WS`.

Rule 2. *A transaction commits under snapshot isolation if it has no WAW-dependencies on any earlier transaction.*

As a result, running all transactions as described in Rule 2 is equivalent to running all transactions concurrently under snapshot isolation in conventional OLTP databases. Note that the result of the database is deterministic as well when Aria runs under snapshot isolation. We omit a formal proof since it is similar to the one presented in Section 5.4.3.

5.4.5 The Phantom Problem

The phantom problem [34] occurs within a transaction when the same query runs more than once but produces different sets of rows. For example, concurrent updates to secondary indexes can make the same scan query observe different results if it runs twice. Serializability does not allow phantom reads. In Aria, secondary indexes are implemented as regular tables that map secondary keys to a set of primary keys. When an insert or a delete affects a secondary index, a reservation must be made in the execution phase. All transactions that access the secondary index are guaranteed to observe the updates to the index by checking RAW-dependencies in the commit

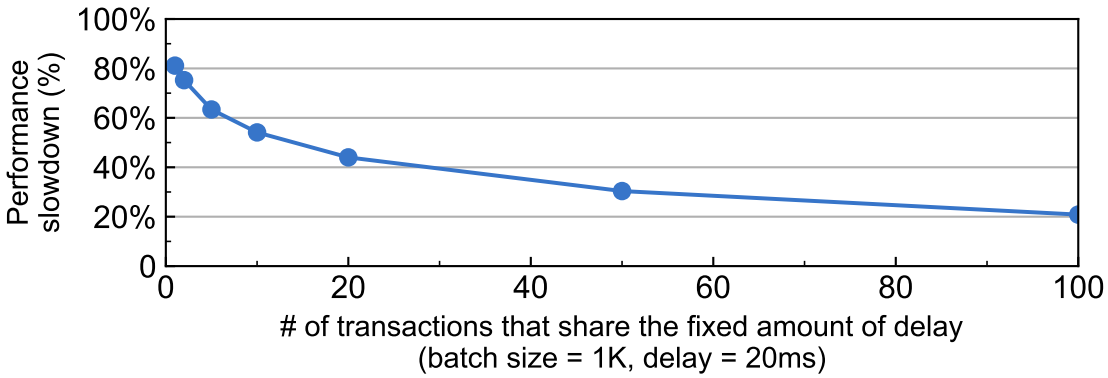


Figure 5-5: Effect of barriers on performance

phase. Therefore, these reservations will cause aborts in the usual way, and and phantoms cannot occur as a result of concurrent reads and inserts/deletes. Phantoms are also possible in range queries (e.g., a table scan). However, standard techniques such as multi-granularity locking [40] can be applied.

5.4.6 Limitations

In Aria, a previous batch of transactions must finish executing before a new batch can begin, since a barrier exists across batches. The system could have undesirable performance due to imbalanced workloads among transactions. Note that because we use round-robin scheduling and execute many transactions in a batch, unless there are very long running transactions, skew in transaction runtimes does not significantly affect the overall runtime of a transaction.

We ran a YCSB workload (see Section 5.8 for more details) to study the effect of imbalanced workloads on our current implementation. In this experiment, we uniformly distribute a fixed total amount of 20 ms delay across multiple transactions in a batch. For example, each transaction has a delay of 2 ms if the total amount of delay is distributed across ten transactions. In Figure 5-5, we plot the system's performance slowdown when the delay is distributed across different numbers of transactions. The maximum throughput is measured when all transactions share the fixed amount of delay (i.e., no stragglers or every transaction is a "straggler"). The results with different fixed total amount of delay follow the same trend and are not shown for clarity.

In the extreme scenario when only a single transaction has a 20 ms delay (i.e., a single straggler), the slowdown is about 81%. As the delay is distributed across more transactions (i.e., more stragglers but each one has shorter delay), the slowdown goes down. For example, when the delay is distributed across more than 100 transactions, the slowdown is less than 20%.

5.5 Deterministic Reordering

The ordering of input transactions affects which transactions can commit within a batch. In this section, we first introduce a motivating example, and then present the deterministic reordering algorithm, which reduces aborts by transforming RAW-dependencies to WAR-dependencies. Finally, we prove its correctness and give a theoretical analysis of its effectiveness. In Section 5.6, we will discuss a fallback strategy that Aria adopts when a workload suffers from a high abort rate due to WAW-dependencies.

5.5.1 A Motivating Example

We use an example to show how the ordering of transactions changes which and how many transactions can commit.

Example 2. *Consider the following three transactions:* T_1*:* $y = x$*,* T_2*:* $z = y$*, and* T_3*: Print* $y + z$

As shown on the top of Figure 5-6, transaction T_2 has a RAW-dependency on transaction T_1, since transaction T_2 reads record y after transaction T_1's write. Following Rule 1, only transaction T_1 can commit, since both transaction T_2 and T_3 have RAW-dependencies. However, if we commit all three transactions, the result is still serializable with an equivalent serial order: $T_3 \rightarrow T_2 \rightarrow T_1$.

The key insight is that by reordering some transactions, RAW-dependencies can be transformed to WAR-dependencies, allowing more transactions to commit. This is because our system does not require aborts as a result of WAR-dependencies.

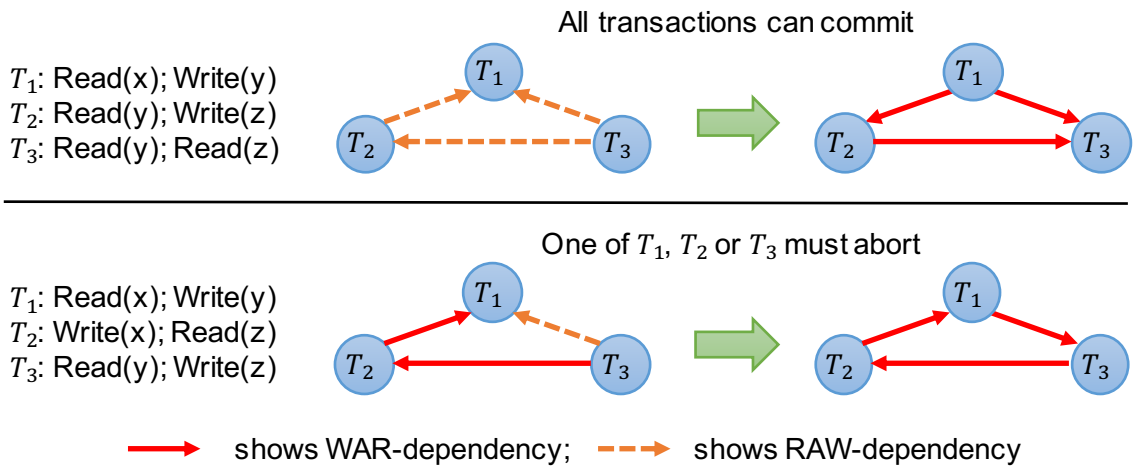


Figure 5-6: Illustrating deterministic reordering

5.5.2 The Reordering Algorithm

By slightly modifying Rule 1, the reordering algorithm can commit more transactions within a batch.

Rule 3. *A transaction commits if two conditions are met: (1) it has no WAW-dependencies on any earlier transaction, and (2) it does not have both WAR-dependencies and RAW-dependencies on earlier transactions (those with smaller TIDs).*

The pseudocode for this algorithm is given in Figure 5-7. Lines 5–10 encode the checking of Rule 3. This code replaces the Commit function in Figure 5-4. Like the algorithm presented in the previous section, this code can be run in parallel, allowing each transaction to commit or abort based on its read/write set and the reservation tables alone. Because this algorithm allows transactions with RAW-dependencies to commit as long as they have no WAR-dependencies, it can result in a serial order that is different from the input order. For example in Example 2 (shown on the top of Figure 5-6), transaction T_2 and T_3 have RAW-dependencies, but there are no WAR-dependencies, and all three transactions can commit with an equivalent order $T_3 \rightarrow T_2 \rightarrow T_1$.

WAW and RAW-dependencies are detected using the write reservation table as described in Section 5.4. To support WAR-dependency detection, the system also

needs to make read reservations in the execution phase as shown from line 1 to line 3 in Figure 5-7. With the read reservation table, for each record in the database, the system knows which transaction first read it – i.e., the one with the smallest TID. In the commit phase, a WAR-dependency will be detected if any key in a transaction's write set appears in the read reservation table and the read is from a different transaction.

Read-only transactions: According to Rule 3, read-only transactions can commit directly. This is because there are no WAW dependencies or WAR dependencies. Intuitively, it is equivalent to running all read-only transactions at the beginning of a batch, since they do not modify the database. However, since Aria makes reservations for a transaction's reads as discussed earlier, these read reservations would introduce unnecessary aborts (i.e., due to spurious WAR-dependencies) for other transactions. As a result, Aria does not make read reservations for read-only transactions.

5.5.3 Proof of Correctness

We now show the correctness of our reordering algorithm.

Theorem 2. *Aria following Rule 3 enforces determinism and serializability.*

We next present two definitions for dependency graphs before presenting the proof of Theorem 2.

Definition 1. *Dependency graph: a directed graph whose vertices are transactions, with edges between transactions with RAW-dependencies and WAR-dependencies, as shown in the left side of Figure 5-6.*

For ease of presentation, we do not consider transactions that have WAW-dependencies on earlier transactions. They are always aborted in Aria and will be re-executed in the next batch. An edge from transaction T_j to T_i indicates that T_j performed the conflicting operation after T_i. This is because conflicts are initially resolved in the order of input transactions for all edges in the graph.

As shown in the right side of Figure 5-6, RAW-dependencies can be transformed to WAR-dependencies by reversing the direction of edges in the transformed dependency

graph. Reversing a RAW edge from transaction T_j to T_i effectively moves T_j before T_i in the serial order, and thus creates a WAR-dependency from T_i to T_j. Based on this, we define the transformed dependency graph.

Definition 2. *Transformed dependency graph: a dependency graph with all RAW-dependencies transformed to WAR-dependencies.*

However, not all transactions can commit even with reordering, since a topological ordering may not exist if there are cycles in the transformed dependency graph.

Example 3. *Consider the following three transactions:* T_1*:* $y = x$*,* T_2*:* $x = z$*, and* T_3*:* $z = y$

As shown on the bottom of Figure 5-6, the transformed dependency graph is not cycle free, and one of T_1, T_2 or T_3 must abort to achieve serializability. To commit as many transactions as possible, we have to remove as few vertices (i.e., abort as few transactions) as possible from the transformed dependency graph to make it acyclic. This problem is well-known as the *feedback vertex set* problem[1] and is shown NP-Complete [45].

Next, we show Aria effectively makes the transformed dependency graph acyclic by applying Rule 3 with Lemma 1, which will be used to prove Theorem 2.

Lemma 1. *The transformed dependency graph is acyclic after applying Rule 3.*

Proof. (BY CONTRADICTION.) Assume the transformed dependency graph is not acyclic. Hence, there must be a sequence of transactions that forms a cycle, i.e., $T_i \to T_j \to \cdots \to T_k \to T_i$. Without loss of generality, we assume $i > j$ and $i > k$. In the transformed dependency graph, there is a WAR-dependency from transaction T_k to T_i. Since $i > k$, this WAR-dependency must be a RAW-dependency from transaction T_i to T_k in the original dependency graph. This is because dependencies only exist from transactions with larger TIDs to those with smaller TIDs. Similarly, the WAR-dependency from transaction T_i to T_j must be a WAR-dependency in the

[1] A feedback vertex set of a graph is a set of vertices whose removal leaves a graph without cycles.

```
1  Function: ReserveRead(T, reads)
2        for each key in T.RS:
3           reads[key] = min(reads[key], T.TID)
4
5  Function: Commit(T, db, reads, writes)
6        # T aborts if writes are not installed
7        if WAW(T, writes):
8           return
9        if WAR(T, reads) == false or RAW(T, writes) == false:
10          Install(T, db)
11
12 Function: WAW(T, writes) = HasConflicts(T.TID, T.WS, writes)
13 Function: WAR(T, reads) = HasConflicts(T.TID, T.WS, reads)
14 Function: RAW(T, writes) = HasConflicts(T.TID, T.RS, writes)
```

Figure 5-7: Pseudocode for deterministic reordering

original dependency graph. Hence, transaction T_i has both WAR-dependency to T_j and RAW-dependency to T_k simultaneously. According to Rule 3, transaction T_i does not exist, which is a contradiction. □

We now give the proof to Theorem 2 by showing an acyclic transformed dependency graph indicates that the transactions can commit with a serializable schedule.

Proof. If no cycles exist in the transformed dependency graph, there exists a topological ordering $\mathcal{O}$ such that for every WAR-dependency from transaction T_i to T_j, we have $\mathcal{O}(T_i) < \mathcal{O}(T_j)$. In addition, for any transaction T_i, its read set does not overlap with any earlier transaction's write set. This is because only WAR-dependencies exist in the transformed dependency graph. Hence, the serializable schedule is the same as the topological ordering $\mathcal{O}$ and the effect of executing these transactions in parallel is identical to that running them serially following the topological ordering $\mathcal{O}$. □

For example, the serializable schedule of the transformed dependency graph on the top of Figure 5-6 is $T_1 \rightarrow T_2 \rightarrow T_3$, which is the same as the topological ordering. In contrast, there is no equivalent serializable schedule to the transformed dependency graph on the bottom of Figure 5-6, since it is not acyclic.

5.5.4 Effectiveness Analysis

Suppose we have a table with $\mathcal{T}$ records. Let $\mathcal{R}$ and $\mathcal{W}$ be the number of read and write operations in a transaction and assume each data access (i.e., a read or a write operation) is independent and follows a uniform distribution.

We first analyze the probability that an operation does not access to a record that has been updated by earlier transactions. If the operation is from transaction T_i (numbered from 0), we have

$$\Pr(\text{an operation has no conflicts}) = (1 - \frac{\mathcal{W}}{\mathcal{T}})^i$$

and therefore,

$$\Pr(\neg \text{ WAW-dependencies exist}) = (1 - \frac{\mathcal{W}}{\mathcal{T}})^{i\mathcal{W}}$$
$$\Pr(\neg \text{ RAW-dependencies exist}) = (1 - \frac{\mathcal{W}}{\mathcal{T}})^{i\mathcal{R}}$$

likewise, we have

$$\Pr(\neg \text{ WAR-dependencies exist}) = (1 - \frac{\mathcal{R}}{\mathcal{T}})^{i\mathcal{W}}$$

since WAR-dependencies depend on the number of read operations from earlier transactions.

According to Rule 1, transaction T_i can commit if it does not have any WAW-dependencies or RAW-dependencies:

$$\begin{aligned}\Pr(T_i \text{ commits}) \quad = \quad & \Pr(\neg \text{ WAW-dependencies exist}) * \\ & \Pr(\neg \text{ RAW-dependencies exist})\end{aligned}$$

The commit rule can be relaxed by Rule 3. With deterministic reordering, we have

$$\begin{aligned} & \Pr(T_i \text{ commits with deterministic reordering}) \\ = \; & \Pr(\neg \text{ WAW-dependencies exist}) * \big\{1 \\ & -\big[1 - \Pr(\neg \text{ WAR-dependencies exist})\big] \\ & *\big[1 - \Pr(\neg \text{ RAW-dependencies exist})\big]\big\} \end{aligned}$$

For the sake of completeness, we give the probability that a transaction can commit under snapshot isolation (SI) according to Rule 2 as well:

$$\Pr(T_i \text{ commits under SI}) = \Pr(\neg \text{ WAW-dependencies exist})$$

where read-write conflicts are not detected.

Our deterministic reordering algorithm allows each transaction to execute concurrently without coordination, even though a serial but more expensive reordering algorithm could potentially produce a better schedule to reduce more aborts. Note that aborting more transactions than are strictly needed to improve performance is a tried and true technique: i.e., using a coarser granularity of locking creates false sharing, resulting in unnecessary aborts, but can improve performance as fewer locks need to be tracked.

5.6 The Fallback Strategy

Because Aria does not need to determine read/write sets up front, and can run in parallel on multiple threads within a replica, it provides superior performance to existing deterministic databases when there are few RAW-dependencies and WAW-dependencies between transactions in a batch. The deterministic reordering optimization can transform RAW-dependencies to WAR-dependencies, allowing many of conflicting transactions to commit under serializability.

In this section, we discuss a fallback strategy that Aria adopts when a workload

suffers from a high abort rate due to WAW-dependencies, e.g., two transactions update the same record. The key idea is that we add a new fallback phase at the end of the commit phase and re-run aborted transactions following the dependencies between conflicting transactions. As discussed in Section 5.3, each transaction reads from the current snapshot of the database in the execution phase and then decides if it can commit in the commit phase. The observation here is that each transaction knows its complete read/write set when the commit phase finishes. The read/write set of each transaction can be used to analyze the dependencies between conflicting transactions. Therefore, many conflicting transactions do not need to be re-scheduled in the next batch. Instead, Aria can employ the same mechanism as in existing deterministic databases to re-run those conflicting transactions.

In our current implementation, we use an approach similar to Calvin, which naturally supports distributed transactions, although any deterministic concurrency control is potentially feasible. In the fallback phase, some worker threads work as lock managers to grant read/write locks to each transaction following the TID order. A transaction is assigned to one of the available worker threads, and executes against the current database (i.e., with changes made by transactions with smaller TIDs) once it obtains all locks. After the transaction commits, it releases all locks. As long as the read/write set of the transaction does not change, the transaction can still commit deterministically. Otherwise, it will be scheduled and executed in the next batch in a deterministic fashion.

The original design of Calvin uses only one thread to grant locks in the predetermined order. However, on a modern multicore node, this single-threaded lock manager cannot saturate all available workers [81]. To better utilize more CPU resources, our implementation uses multiple lock manager threads to grant locks. Each lock manager is responsible for a different portion of the database. Our new design is logically equivalent to but more efficient than the original design [97], which ran multiple Calvin instances on a single node. This is because worker threads now communicate with each other through shared memory rather than sockets.

A moving average of the abort rate in a workload is monitored by the sequencing

layer. When the abort rate exceeds the threshold, the sequencing layer deterministically switches the system to using the fallback, and then turns off the fallback when the abort rate drops. Note that the fallback strategy guarantees that Aria is at least as effective as existing deterministic databases in the worst case, since non-conflicting transactions only run once. However, existing deterministic databases run input transactions at least twice: once to determine the read/write set, and once to execute the query.

5.7 Implementation

We now describe the data structures used to implement Aria and how the system supports distributed transactions and provides fault tolerance across multiple replicas.

5.7.1 Data Structures

Aria is a distributed in-memory deterministic OLTP database. It provides a relational model, where each table has a pre-defined schema with typed and named attributes. Clients interact with the system by calling pre-compiled *stored procedures* with different parameters, as in many popular systems [97, 98, 105]. A stored procedure is implemented in C++, in which arbitrary logic (e.g., read/write operations) is supported. The system targets one-shot and short-lived transactions and does not support multi-round and interactive transactions. It does not provide a SQL interface.

Each table has a primary key and some number of attributes. Tables are currently implemented as a primary hash table and zero or more secondary hash tables as indexes, meaning that range queries are not supported. This could be easily adapted to tree structures [11, 62, 103]. A record is accessed via probing the primary hash table. For secondary lookups, two probes are needed, one in a secondary hash table to find the primary key of a record, followed by a lookup in the primary hash table.

As discussed in previous sections, a transaction makes reservations for reads and writes in the execution phase, which are used for conflict detection in the commit phase. Essentially, these reservations are collections of key-value pairs. The key is the

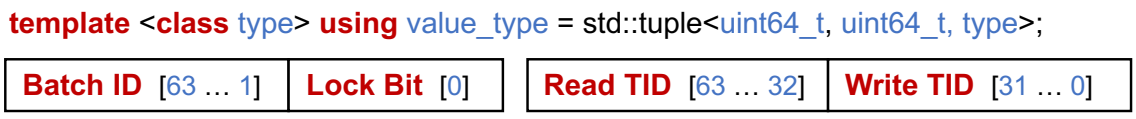


Figure 5-8: Per-record metadata format in Aria

primary key of the table that a transaction accesses and the value is the transaction's TID. Since these key-value pairs are table specific, we record the TID value along with the batch ID with two 64-bit integers and attach them to each record in the table's primary hash table, as shown on the top of Figure 5-8.

These two 64-bit integers are used to keep track of reservations. The format is shown on the bottom of Figure 5-8. The first 64-bit integer (except the lowest bit) is used to keep track of the batch ID that a reservation belongs to. The second 64-bit integer — 32 bits for read and 32 bits for write — is used to keep track of the TID of the transaction that makes the reservation. If a read or a write reservation comes with a larger batch ID, both integers will be overwritten directly. For example, a read reservation with a larger batch ID will update the batch ID, the read TID, and set the write TID to zero. Note that we use zero to indicate a record has not been read or written. Otherwise, the read TID or the write TID is compared with the transaction's TID. If the transaction's TID is smaller, meaning it appears earlier in the batch, the corresponding read TID or write TID is updated. Since the per-record metadata is maintained in two 64-bit integers, we use the lowest bit of the first integer as a lock bit to isolate concurrent updates between multiple worker threads.

5.7.2 Distributed Transactions

Aria supports distributed transactions through partitioning. Each table is horizontally partitioned across all nodes and each partition is randomly hashed to one node. Note that the sequencing layer is allowed to send transactions to any node for execution. However, if a transaction was sent to the node having most data it accesses, network communication could be reduced. In real scenarios, the partitions that a transaction accesses can be inferred from the parameters of a stored procedure in

some cases (e.g., the `New-Order` transaction on TPC-C). Otherwise, the transaction can be send to any node in Aria for execution.

As discussed earlier, the execution phase and the commit phase are separated with barriers. In the distributed setting, these barriers are implemented through network round trips. For example, a designated coordinator (selected from among all nodes) decides that the system can enter the commit phase once all nodes finish the execution phase.

In the execution phase, a remote read request is sent, when a transaction reads a record that does not exist on the local node. Similarly, read/write reservation requests are also sent to the corresponding nodes. In the commit phase, network communication is also needed to detect conflicts among transactions. Note that worker threads within the same node do not communicate by exchanging messages. Instead, all operations are conducted directly in main memory. Aria batches these requests to reduce latency and improve the network's performance. Once a transaction commits, write requests are also batched and sent to remote nodes to commit changes to the database.

5.7.3 Fault Tolerance

Fault tolerance is significantly simplified in a deterministic database. This is because there is no need to maintain physical undo/redo logging, instead, the system only needs to make input transactions durable in the sequencing layer. In addition, replicas do not communicate with one another, meaning global barriers do not exist across replicas. We now discuss when the result of a transaction can be released to the client and how checkpointing works.

Every transaction first goes to the sequencing layer, in which it is assigned a batch ID and a transaction ID. The sequencing layer next forwards the same input transactions to all replicas. We next discuss how the sequencing layer communicates with one replica, since the mechanism is the same for all replicas. Before the execution phase begins, each node within a replica will receive a different portion of a batch of transactions. Once all transactions finish execution, the replica communicates with

the sequencing layer again, notifying it the result of each transaction. If a transaction aborts due to conflicts, the sequencing layer will put the transaction into the next batch.

The result of a committed transaction can be released to the client as soon as it commits in the commit phase. This is because its input was durable before execution. Once a failure occurs, every transaction will run again and commit in the same order due to the nature of deterministic execution.

To bound the recovery time, the system can be configured to periodically checkpoint the database to disk. During a checkpoint, a replica stops processing the transactions and saves a consistent snapshot of the database to disk. As long as there is more than one replica, clients are not aware of the pause when checkpoints occur, since each replica can checkpoint independently and transactions are always run by the system through other available replicas. On recovery, a replica loads the latest checkpoint from disk to the database and replays all transactions since the checkpoint.

5.8 Evaluation

In this section, we evaluate the performance of Aria focusing on the following key questions:

- How does Aria perform compared to conventional deterministic databases and primary-backup databases?
- What is the scheduling overhead of Aria?
- How effective is the deterministic reordering?
- How does batch size affect throughput and latency?
- How does Aria scale?

5.8.1 Experimental Setup

We ran our experiments on a cluster of `m5.4xlarge` nodes running on Amazon EC2 [2]. Each node has 16 2.50 GHz virtual CPUs and 64 GB RAM running 64-bit Ubuntu

18.04 with Linux kernel 4.15.0 and GCC 7.3.0. The nodes are connected with a 10 GigE network.

To avoid an apples-to-oranges comparison, we implemented the following concurrency control protocols in C++ in the same framework: (1) **Aria:** our algorithm with deterministic reordering. The fallback strategy is disabled, (2) **AriaFB:** different from Aria, AriaFB always uses the fallback strategy to re-run aborted transactions, (3) **BOHM:** a single-node MVCC deterministic database using dependency graphs to enforce determinism [36], (4) **PWV**: a single-node deterministic database that enables early write visibility [35], (5) **Calvin:** a distributed deterministic database using ordered locks to enforce determinism [97], and (6) **PB:** a conventional non-deterministic primary-backup replicated database.

Our implementation of Calvin has the same optimization as in the fallback phase of Aria and we use Calvin-x to denote the number of lock manager threads. The concurrency control protocol used in PB is strict two-phase locking (S2PL). We use a NO_WAIT deadlock prevention strategy, which previous work has shown to be the most scalable protocol [42]. Transactions involving multiple nodes are committed with two-phase commit [65]. PB uses synchronous replication to prevent loss of durability in high-throughput environments. In synchronous replication, locks on the primary are not released until the writes are replicated on the backup. A transaction commits only after the writes are acknowledged by backups.

We use two popular benchmarks, YCSB [18] and a subset of TPC-C [1], to study the performance of Aria. These benchmarks were selected because they are all key-value benchmarks where it is possible for existing deterministic databases to pre-compute read/write sets. Many other workloads cannot easily run on them due to the need for this pre-computation. The Yahoo! Cloud Serving Benchmark (YCSB) is designed to evaluate the performance of key-value systems. There is a single table with ten columns. The primary key of the table is a 64-bit integer and each column has ten random bytes. To make it a transactional benchmark, we wrap operations within transactions [102, 108]. By default, each transaction has 10 operations. The TPC-C benchmark models an order processing application, in which customers place

orders in one of ten districts within a local warehouse. We support the `New-Order` and the `Payment` transaction in this benchmark. 88% of the standard mix consists of these two transactions. The other three transactions require range scans, which are currently not supported. By default, there are 10% `New-Order` and 15% `Payment` transactions that access multiple warehouses. On average, one `NewOrder` transaction is followed by one `Payment` transaction.

We run 12 worker threads and 2 threads for network communication on each node. Note that, in Calvin-x, there are x worker threads granting locks and 12-x worker threads running transactions. By default, we set the number of partitions equal to the total number of worker threads in the cluster, meaning there are 12 partitions on each node. This is because existing deterministic databases [35, 36, 97] require the use of partitionings to use multiple worker threads for better performance. In YCSB, the number of keys per partition is 40K. In TPC-C, the database is partitioned by warehouse and the number of partitions is equal to the number of warehouses.

There is no sequencing layer in our experiments. This is because the sequencing layer is the same in all deterministic databases, and adding the sequencing layer or not does not affect our experimental conclusions. To measure the maximum performance that each system is able to achieve, we use a built-in deterministic workload generator to get an ordered batch of transactions on each node. Transactions are run under serializability. All reported results are the average of ten runs.

We only measure the performance of Aria, AriaFB, BOHM, PWV, and Calvin at one replica, since all replicas achieve comparable performance. We consider two variations in our experiments: (1) a single-node setting, used in from Section 5.8.2 to Section 5.8.5, and (2) a multi-node setting (i.e., a cluster of eight nodes), used in Section 5.8.6 and 5.8.8. In all deterministic databases, we set the batch size to 1K on YCSB and 500 on TPC-C in the single-node setting, and 80K on YCSB and and 4K on TPC-C in the multi-node settings. These parameters ensure the database to achieve the best performance. For PB, a dedicated backup node stands by for replication in the single-node setting. In the multi-node setting, each partition has a primary partition and a backup partition, which are always assigned to two different

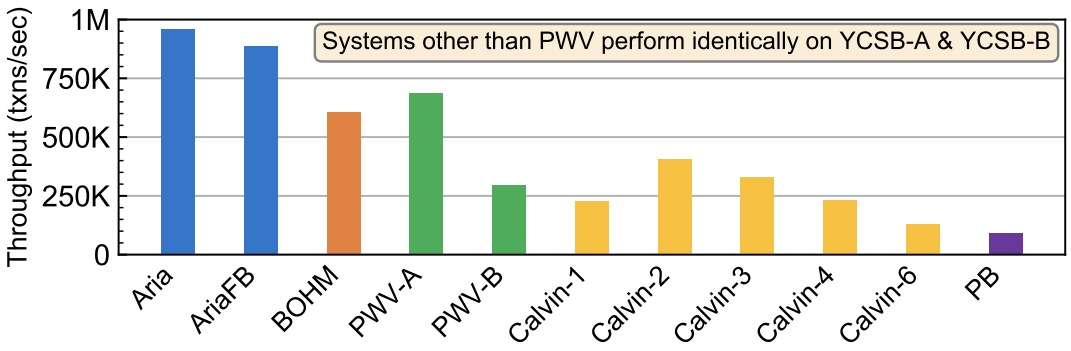


Figure 5-9: Performance on YCSB-A and YCSB-B

nodes.

5.8.2 YCSB Results

We first study the performance of each system on the YCSB benchmark. We run a workload mix of 80/20, meaning each access in a transaction has an 80% probability of being a read operation and a 20% probability of being an update operation. All accesses follow a uniform distribution, which makes most transactions multi-partition transactions.

We use two variants of YCSB: (1) YCSB-A: each read/write is independent, and (2) YCSB-B: the value of a write depends on all reads. Note that only PWV is sensitive to the two variations above, which affect when the writes of a transaction are visible to other transactions. In contrast, conflict detection or dependency analysis stays the same for Aria, AriaFB, BOHM and Calvin. Likewise, PB always acquires all read/write locks before execution and only releases them when a transaction commits. In this experiment, we use PWV-A and PWV-B to denote the performance of PWV on the two variations above respectively. For other systems, the results on YCSB-A and YCSB-B are the same and we only report the result on YCSB-A.

We report the results in Figure 5-9. AriaFB has a slightly lower throughput than Aria, since the abort rate in this workload is about 0.4% and the overhead of the fallback strategy exceeds its benefits. PWV on YCSB-A performs the best among all baselines, since each read/write is independent and it allows early write visibility. In

contrast, PWV has much lower throughput on YCSB-B, since a write is only visible to other transactions when all reads are finished. BOHM has the second highest throughput, but the overhead of managing multiple versions of each record makes it achieve only two thirds of Aria's throughput. The performance of Calvin depends on the number of threads used for granting locks. One lock manager fails to saturate the system. Calvin achieves the best performance when two lock managers are used in this workload. PB's throughput is about 11% of Aria's throughput. This is because one network round trip is needed in PB for each transaction with write operations due to synchronous replication. In our test environment, the round trip time is about 100 μs.

Overall, Aria achieves 1.1x, 1.6x, 1.4x, 3.3x, 2.4x and 10.3x higher throughput than AriaFB, BOHM, PWV-A, PWV-B, Calvin and PB on YCSB, respectively.

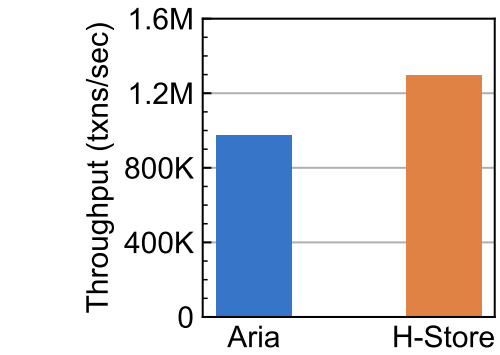
5.8.3 Scheduling Overhead

We now study the scheduling overhead of each concurrency control algorithm. In this experiment, we use the same YCSB workload as in Section 5.8.2, but all transactions access only one partition. This is because H-Store [91] has minimal overhead in concurrency control when transactions do not span partitions.

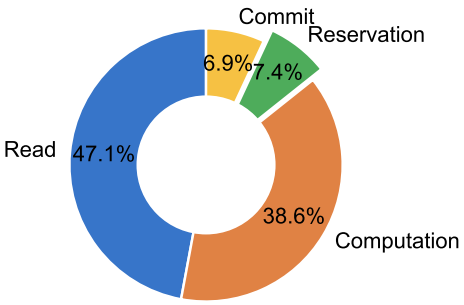
We first compare Aria with our implementation of H-Store in the same framework, which can be used to estimate the upper bound of the transaction throughput that a workload can achieve. We report the result in Figure 5-10(a) and observe that Aria achieves about 75% of H-Store's throughput.

Second, we show a performance breakdown on Aria in Figure 5-10(b). There are three steps in the execution phase, namely, (1) reading the database snapshot, (2) computing the transaction logic, and (3) making reservations. Note that Aria uses read/write reservations to achieve determinism and serializability. Therefore, we consider the third step to be the scheduling overhead, which is about 7.4%.

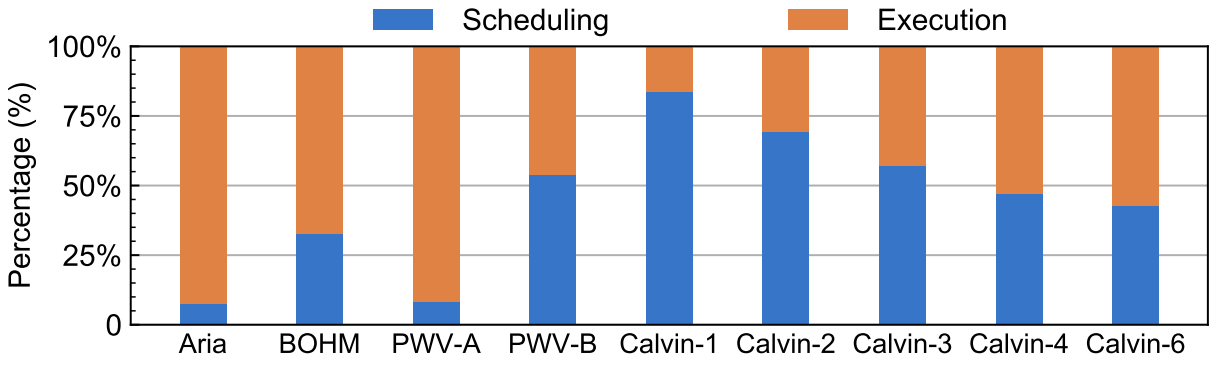
Third, we show the scheduling overhead of each deterministic concurrency control in Figure 5-10(c). Existing deterministic databases run transactions following a dependency graph, meaning there exists a scheduling overhead in each algorithm.



(a) Aria vs. H-Store



(b) Performance breakdown



(c) Scheduling vs. Execution

Figure 5-10: A study of scheduling overhead on each system on YCSB

(e.g., the waiting time wasted in each worker thread). In addition, we consider the time spent on each lock manager in Calvin and the time spent on dependency graph construction in BOHM and PWV to be scheduling overhead. The time spent on placeholder insertion and garbage collection in BOHM is also considered to be scheduling overhead. Among all algorithms, Aria and PWV-A have the lowest scheduling overhead, which partially explains why they have the best performance in Figure 5-9.

5.8.4 Effectiveness of Deterministic Reordering

In this section, we study the effectiveness of the deterministic reordering technique introduced in Section 5.5. There are many factors that influence the effectiveness of this technique, such as the percentage of read-only transactions, access distribution, the database size, and the read/write ratio. In this experiment, we ran the same workload as in Section 5.8.2, but varied the skew factor [41] from 0 (i.e. uniform

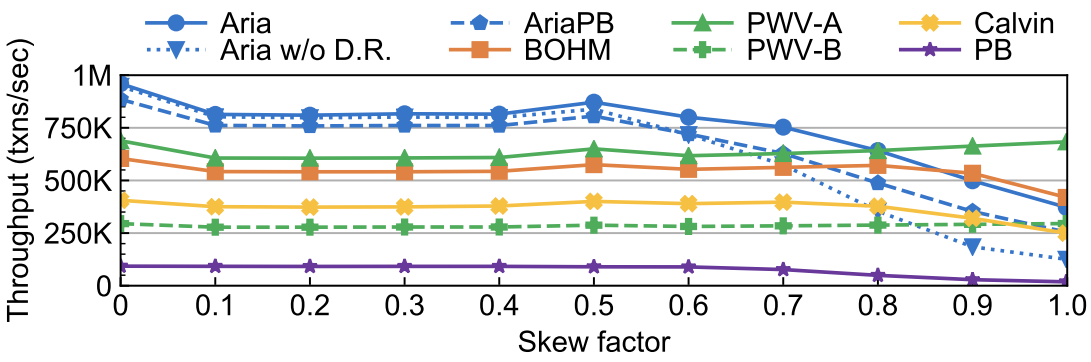


Figure 5-11: Effectiveness of deterministic reordering

distribution) to 0.999 in the workload. Aria w/o D.R. shows the performance of Aria without deterministic reordering. For clarity, in this experiment and all following experiments, we only report the performance of Calvin with the optimal number of lock managers.

We first study the effect of the skew factor on Aria. As shown in Figure 5-11, the throughput goes down when the we increase the skew factor. For example, when the skew factor is 0.999, Aria achieves only 39% of its original throughput when there is no skew. This is because a larger skew factor leads to a higher abort rate due to conflicts and many transactions have to be re-run multiple times. Similarly, a transaction has more difficulty getting read/write locks in PB as the skew factor increases. For this reason, the throughput of PB is only about 20% of the original throughput when there is no skew. BOHM and Calvin also have lower throughput when the skew factor is larger due to less parallelism. However, PWV is less sensitive to the skew factor due to its early write visibility mechanism. The throughput of Aria is lower than PWV-A and BOHM when the skew factor is larger than 0.8 and 0.9 respectively.

We also report the result of AriaFB in this experiment. When the skew factor is small, AriaFB achieves slightly lower throughput than Aria, since it's more efficient to simply retry the aborts in the next batch than re-running the aborts with the fallback strategy. As we increase the skew factor, AriaFB achieves higher throughput than Aria w/o D.R., but it is still slower than Aria, showing the effectiveness of our reordering technique. Under high skew, AriaFB achieves similar throughput to

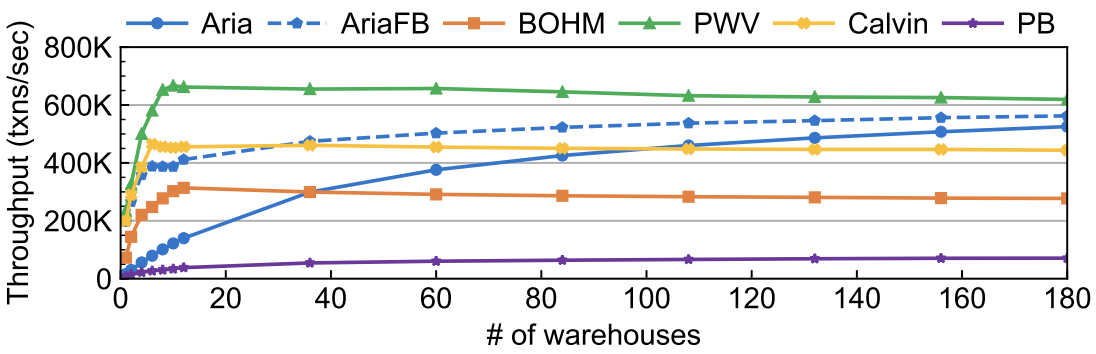


Figure 5-12: Performance on TPC-C

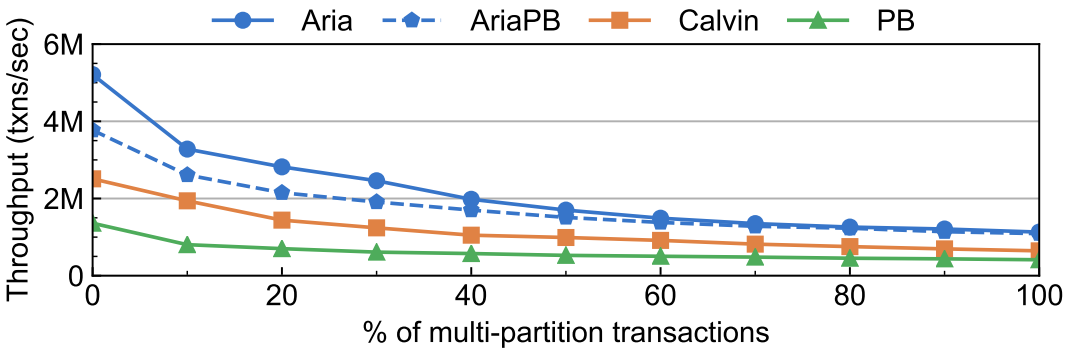
Calvin, since most transactions are executed using the fallback strategy.

We now study how much performance gain Aria can achieve with deterministic reordering. When each access follows a uniform distribution, deterministic reordering achieves similar throughput to the one without deterministic reordering. This is because there are few conflicts and the number of aborts is low. Note that the deterministic reordering mechanism does not slow down the system even though when the contention is low. As we increase the skew factor in the workload, more conflicts exist and Aria with deterministic reordering achieves significantly higher throughput. For example, when the skew factor is 0.999, Aria achieves 3.0x higher throughput than Aria w/o D.R. .

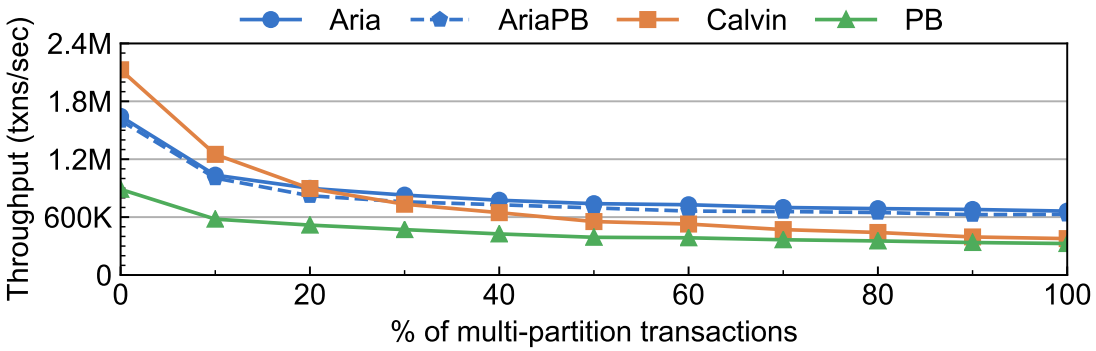
Overall, the deterministic reordering technique is effective to reduce aborts due to RAW-dependencies, making Aria achieve higher throughput than AriaFB on YCSB.

5.8.5 TPC-C Results

We next study the performance of each system on TPC-C. In this experiment, we varied the number of partitions (i.e., the number of warehouses) from 1 to 180. As shown in Figure 5-12, all other deterministic databases have stable performance when the number of partitions exceeds 12, since the maximum parallelism is achieved when each thread has one partition on TPC-C. The throughput of Aria goes up as more partitions exist due to fewer conflicts. When the number of partitions is 180, Aria achieves 85% of the throughput of PWV.



(a) YCSB - batch size: 10K * 8 = 80K



(b) TPC-C - batch size: 500 * 8 = 4K

Figure 5-13: Performance of each system on YCSB and TPC-C in the multi-node setting

Note that there are contended writes in TPC-C, i.e., writes made on the `d_ytd` field in the district table, making Aria perform worse when there are a small number of warehouses. However, AriaFB can commit non-conflicting transactions in the normal commit phase and re-run conflicting transactions with the fallback strategy, making it achieve the best of both worlds. Therefore, the throughput of AriaFB always exceeds both Aria and Calvin on TPC-C.

In summary, with 180 partitions, Aria achieves 1.9x, 1.2x and 7.4x higher throughput than BOHM, Calvin and PB respectively and has comparable throughput to PWV, which enables early write visibility but requires complex transaction decomposition.

5.8.6 Results on Distributed Transactions

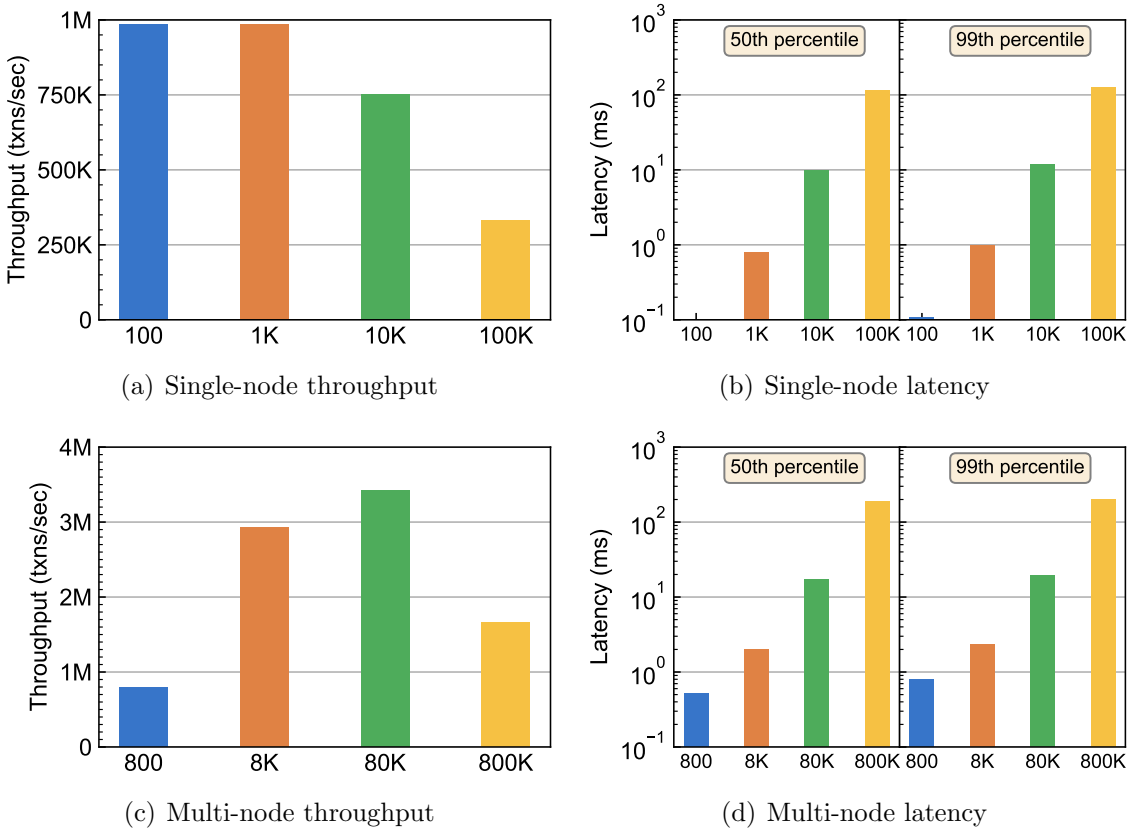
In this section, we study the performance of Aria, AriaFB, Calvin, and PB in a multi-node setting using 8 Amazon EC2 [2] instances. We omit the results of BOHM and PWV, which are designed to be single-node deterministic databases.

We first ran a YCSB workload with a varying percentage of multi-partition transactions and report the results in Figure 5-13(a). In this experiment, a multi-partition transaction only accesses two partitions. Different from the single-node setting, the throughput of all approaches goes down with more multi-partition transactions. This is because more network communication occurs. In addition, AriaFB has lower throughput than Aria, since there are two more barriers per batch when the fallback strategy is employed. In summary, Aria achieves up to 1.4x higher throughput than AriaFB, 1.6x - 2.1x higher throughput than Calvin and 2.7x - 4.1x higher throughput than PB on YCSB.

We also ran a workload of TPC-C and report the result in Figure 5-13(b). We set the number of partitions (i.e., the number of warehouses) to 108 per node, meaning there are 864 partitions in total. Aria achieves higher throughput than Calvin when more multi-partition transactions exist. This is because more work needs to be done in multi-partition transactions in Calvin as discussed in Section 5.2. In addition, Aria and AriaFB achieve similar throughput in this workload, since the performance gain from the fallback strategy is about the same as the overhead of two additional barriers. Overall, Aria achieves up to 1.7x higher throughput than Calvin and up to 2x higher throughput than PB on TPC-C.

5.8.7 Effect of Batch Size

We next study the impact of batch size on the throughput and the latency on a YCSB workload in both single-node and multi-node settings. In Aria, the batch size determines on average how many transactions abort and must re-run within a batch. A smaller batch size leads to a lower abort rate and lower latency, but may introduce high synchronization cost on barriers. A larger batch size reduces the synchronization



(a) Single-node throughput

(b) Single-node latency

(c) Multi-node throughput

(d) Multi-node latency

Figure 5-14: Throughput and latency of Aria on YCSB in both single-node and multi-node settings

cost but leads to a higher abort rate and higher latency.

In Figure 5-14(a), we show the throughput of Aria in the single-node setting. The throughput goes down as the batch size becomes larger, since more transactions abort and must re-run. Figure 5-14(b) shows the latency of Aria in the single-node setting. Intuitively, a longer latency is expected with a larger batch size.

Figure 5-14(c) and Figure 5-14(d) show the throughput and the latency of Aria in the multi-node setting. In the distributed setting, a smaller batch size does not necessarily guarantee higher throughput due to the cost of synchronization. For example, Aria achieves the highest throughput when the batch size is 80K. The result on latency in the distributed setting is similar to that in the single-node setting, i.e., Aria has longer latency with a larger batch size.

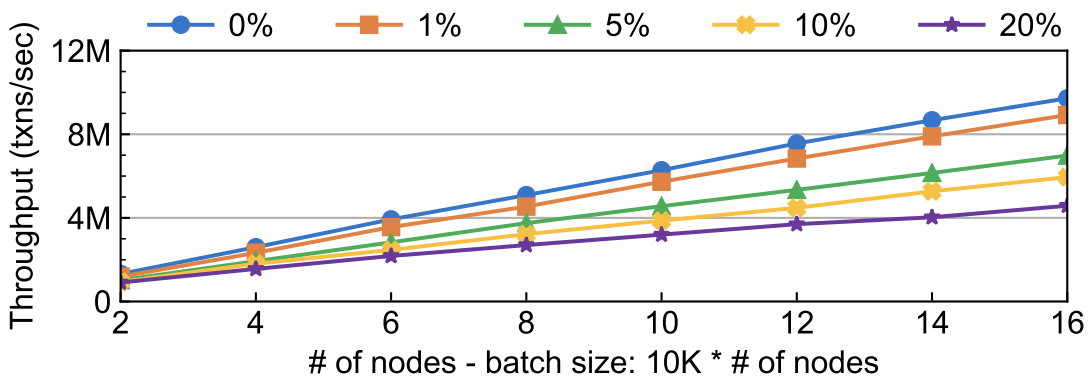


Figure 5-15: Scalability of Aria on YCSB

5.8.8 Scalability Experiment

Finally, we study the scalability of Aria. We ran the same YCSB workload as in Section 5.8.2, but we scale the number of partitions from 24 to 192. The batch size is set to 10K times by the number of nodes. We report the results on five different configurations in Figure 5-15, in which $x\%$ denotes the percentage of multi-partition transactions. Again, a multi-partition transaction accesses only two partitions. We can observe that Aria achieves almost linear scalability in all configurations. For example, it achieves around 6.3x higher throughput with 16 nodes compared to the one with 2 nodes when 10% multi-partition transactions exist.

5.9 Related Work

Aria builds on many pieces of related work for its design, including transaction processing, highly available systems, and deterministic databases.

5.9.1 Transaction Processing

There has been a resurgent interest in transaction processing, as both scale-up multi-core databases [30, 47, 54, 70, 98, 106, 108] and scale-out distributed databases [20, 42, 53, 59, 67, 68, 91] are becoming more popular. Silo [98] avoids shared-memory write by dividing time into short epochs and each thread generates its own timestamp by embedding the global epoch. Aria uses a similar batch-based execution model to run

transactions. Aria deterministically detects conflicts in a separate commit phase after all transactions finish execution. Doppel [70] allows different orderings of updates as long as updates are commutative. Aria can be extended to support commutative updates as well to reduce conflicts between transactions. More recent database systems [16, 32, 104, 105] build on RDMA and HTM. Aria can use RDMA to further decrease the overhead of network communication and barriers across batches.

5.9.2 Highly Available Systems

Modern database systems support high availability, such that an end user does not experience any noticeable downtime when a subset of servers fail. High availability is usually achieved through replication [14, 39, 85, 99], which itself can be either asynchronous [17, 25] or synchronous [46]. Asynchronous replication reduces latency during replication by sacrificing consistency when failures occur. For this reason, synchronous replication which enables strong consistency is becoming more popular [19, 60].

Many recent protocols [78, 92, 101] are developed with high availability in mind. TAPIR [112] builds consistent transactions through inconsistent replication. SiloR [113] uses parallel recovery via value replication, in which logs can be replayed in any order. Most existing work above achieves high availability through shipping the output. Instead, Aria achieves high availability through replicating the input and running transactions deterministically across replicas.

5.9.3 Deterministic Databases

The design of deterministic database systems dates back to the late 1990s [43, 69]. Each replica achieves consistency through a deterministic scheduler that produces exactly the same thread interleaving [43]. As more systems favor full ACID properties, deterministic systems have attracted a great deal of attention recently [4, 37, 82, 83, 84, 96]. Existing deterministic databases [35, 36, 97] require that the read set and write set of a transaction must be known a priori. Hyder [9], FuzzyLog [57] and

Tango [6], unlike conventional deterministic databases [35, 36, 97], allow each node to read the same database via a shared log, which establishes a global order across all updates. However, the result could be different if multiple shared-log systems are deployed independently. Instead, conventional deterministic databases always produce the same result given the same input transactions.

5.10 Summary

In this chapter, we presented Aria, a new distributed and deterministic OLTP database. By employing deterministic execution, Aria is able to efficiently run transactions across different replicas without coordination, and without prior knowledge of read and write sets as in existing work. The system executes a batch of transactions against the same database snapshot in an execution phase, and then chooses deterministically the ones that should commit to ensure serializability in a commit phase. We also propose a novel deterministic reordering mechanism so that Aria commits transactions in an order that reduces the number of conflicts. Our results on two popular benchmarks show that Aria outperforms systems with conventional nondeterministic concurrency control algorithms and state-of-the-art deterministic databases by a large margin on a single node and up to a factor of two on a cluster of eight nodes.

Chapter 6

Discussion

In previous chapter, we presented the design of three different systems we have built to address the inefficiency and limitations of existing distributed databases. In this chapter, we discuss the tradeoffs of conventional designs and our proposed systems, and summarize the key features of each design in Table 6.1.

6.1 Conventional Designs

The most common way to run transactions in distributed databases is to use two-phase commit (2PC). To keep replicas strongly consistent, synchronous replication is used. Although some systems adopt asynchronous replication to achieve higher performance, ACID properties are often sacrificed and we do not consider these systems in this chapter.

Systems based on conventional designs provide almost linear horizontal scalability on partitionable workloads, i.e., higher throughput can be achieved by adding more nodes to the cluster of the database. In addition, such systems also support most types of workload from the perspective of application developers. For example, they provide support for interactive transactions.

In terms of latency and throughput, these systems provide lower commit latency, since each transaction can commit independently. This feature makes such systems a popular choice when workloads demand low commit latency (e.g., real time applica-

Table 6.1: A summary of conventional designs, STAR, COCO and Aria

	Conventional Designs	STAR	COCO	Aria
Scale out	Linear	Sublinear	Linear	Linear
Deterministic	No	No	No	Yes
Write latency	High	Low	Medium	Medium
Commit latency	Low	High	High	High
Replication cost	High	Workload dependent	High	Low
Transaction throughput	Low	High	High	High
Interactive transactions support	Yes	Yes (May hurt performance)	Yes (May hurt performance)	No
Single-node & distributed deployment	Both	Distributed deployment only	Both	Both

tions). However, the throughout is impaired due to 2PC and synchronous replication. For example, a write is always associated with at least one network round trip. In addition, replication is usually achieved through replicating the output of transactions, which consumes much network bandwidth.

6.2 STAR

STAR outperforms databases based on conventional designs with only a few nodes (e.g., less than ten nodes) in a cluster, as long as a single node in the cluster can hold an entire copy of the database. Since STAR has sublinear scalability, we expect that its performance can be exceeded by conventional databases but with much more nodes, e.g., tens of nodes in a cluster.

In addition, due to the phase-switching algorithm, transactions in STAR always write to the local database and apply writes to replicas in an asynchronous way. Depending on the workload, some optimizations are possible during replication (e.g., replicating the operations of transactions rather than values) in the partitioned phase, reducing the pressure on network. STAR can achieve the best of both worlds (i.e., partitioning-based systems and non-partitioned systems) as long as a workload consists of both single-partition transactions and distributed transactions. However, if a workload consists of only single-partition transactions or distributed transactions,

although not very likely, STAR is not a good fit and application developers should choose other designs.

Note that STAR can support interactive transactions but performance may be impaired. This is because a transaction must finish execution before the system switches from one phase to another phase. If an interactive transaction is a long running transaction (e.g., a transaction having increased latency due to multiple round communication), the throughput of STAR will be greatly impacted.

Finally, STAR cannot be deployed on a single node, since it requires two types of replicas in a cluster.

6.3 COCO

COCO outperforms databases based on conventional designs by grouping transactions into epochs and treating epochs as commit units. In this way, the overhead of 2PC and synchronous replication is greatly reduced. It also has linear scalability as in existing databases, making it scale to tens and hundreds of nodes in a cluster.

Since a transaction in COCO requires fewer network round trips, the writes of a transaction can be applied to the database faster than existing databases. Similarly, replication can also perform in an asynchronous way. Such design makes COCO a better fit for workloads that have high contention than conventional databases. This is because locks are now held for shorter period of time, allowing each transaction to have lower latency and eventually making the system have higher throughput.

As in STAR, COCO can support interactive transactions as well, but the commit latency may be impaired. This is because an epoch cannot commit until all transactions belonging to the epoch have committed. If an interactive transaction spans several epochs, the commit latency is increased as well.

Finally, COCO can be deployed in both single node setting and distributed setting.

6.4 Aria

Aria outperforms databases based on conventional designs via deterministic execution, which does not require 2PC. Different from existing deterministic databases [36, 97], it does not require any analysis or pre-execution of input transactions. In addition, there is no need for replicas to communicate when keeping replicas strongly consistent in a deterministic database.

Since transactions run in batches in Aria, all reads and writes are buffered, making the system use the network more efficiently. Aria is a good fit for workloads that have low contention or medium contention due to RAW-dependencies. This is because transactions aborted due to RAW-dependencies can be reordered and commit with a deterministic reordering algorithm. For workloads having high contention and much WAW-dependencies, Aria provides a fallback strategy, making it at least as effective as existing deterministic databases, which do not suffer from high contention or much WAW-dependencies.

Different from STAR and COCO, Aria does not support interactive transactions. This is because the system cannot enter the commit phase until all transactions have finished execution in the execution phase. For the same reason, Aria will have undesirable performance when there exists long-running transactions in a workload, especially, when there is skew in transaction running time in a batch.

Finally, Aria can be deployed in both single node setting and distributed setting.

6.5 Summary

In this thesis, we presented three systems: (1) STAR, (2) COCO, and (3) Aria to address the research challenges in distributed and highly available databases. We built these three systems with the same design principle — managing transaction processing at the granularity of epochs or batches. In STAR, epochs enable more efficient replication through asynchronous replication without sacrificing consistency guarantees. In COCO, epochs amortize the cost of two-phase commit and replication

by committing or aborting a batch of transactions all together. In Aria, a batch of transactions are first run and conflicts between transactions within the batch are next analyzed deterministically. Our experiments show that all three systems can outperform systems with conventional designs by a large margin on two popular benchmarks (YCSB and TPC-C).

Chapter 7

Future Work

In this chapter, we discuss some promising opportunities for future work.

Database administrators (DBAs) play an important role in OLTP databases, since many critical components still rely on them to manually configure and optimize parameters. However, it is hard to implement the optimal collection of indexes given a transactional workload, even for an experienced DBA. To satisfy the latency and throughput requirements of the applications with minimal labor costs and computing resources, an interesting problem to study is how to automatically design the best configuration. In addition, workload characteristics such as data contention often are highly dynamic. Therefore, an initially good configuration does not necessarily work well over time. To tackle these problems, we plan to use machine learning to continuously learn the appropriate configuration based on the workload.

Another interesting problem is how to design data partitioning strategies that make distributed OLTP databases scalable. This inevitably involves distributed transactions that touch several data partitions across multiple nodes, which cause significant performance degradation. To solve this problem, we plan to use reinforcement learning to continuously develop good partitioning strategies, which minimize the number of distributed transactions for a given workload. In addition, it's also interesting to use machine learning to generate contention-aware scheduling strategies to intelligently determine which transactions to run together and in which order. Finally, we will also investigate how to dynamically select from different concurrency

control protocols the one that best fits the current workload.

Chapter 8

Conclusion

In this thesis, we presented the design of three systems, namely, (1) STAR, (2) COCO, and (3) Aria. The three systems above address the inefficiency and limitations of existing distributed OLTP databases while using different mechanisms and bearing various tradeoffs. In particular, STAR eliminates two-phase commit and network communication through asymmetric replication. COCO eliminates two-phase commit and reduces the cost of replication through epoch-based commit and replication. Aria eliminates two-phase commit and the cost of replication through deterministic execution. Our experiments on two popular benchmarks (YCSB and TPC-C) showed that these three systems outperform conventional designs by a large margin.

Bibliography

[1] TPC Benchmark C. http://www.tpc.org/tpcc/, 2010.

[2] Amazon EC2. https://aws.amazon.com/ec2/, 2020.

[3] Google Cloud. https://cloud.google.com/, 2020.

[4] Daniel J. Abadi and Jose M. Faleiro. An overview of deterministic database systems. *Commun. ACM*, 61(9):78–88, 2018.

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

[6] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: distributed data structures over a shared log. In *SOSP*, pages 325–340, 2013.

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

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[10] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Trans. Software Eng.*, 5(3):203–216, 1979.

[11] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: A height optimized trie index for main-memory database systems. In *SIGMOD Conference*, pages 521–534, 2018.

[12] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. Distributed snapshot isolation: Global transactions pay globally, local transactions pay locally. *VLDB J.*, 23(6):987–1011, 2014.

[13] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *SOSP*, pages 1–11, 1995.

[14] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *SIGMOD Conference*, pages 739–752, 2008.

[15] Prima Chairunnanda, Khuzaima Daudjee, and M. Tamer Özsu. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *PVLDB*, 7(11):947–958, 2014.

[16] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *EuroSys*, pages 26:1–26:17, 2016.

[17] 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. *PVLDB*, 1(2):1277–1288, 2008.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. 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. In *OSDI*, pages 261–264, 2012.

[20] James A. Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *ATC*, pages 223–235, 2012.

[21] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, pages 727–743, 2018.

[22] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[23] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.

[24] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[26] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8, 1984.

[27] Akon Dey. Scalable transactions across heterogeneous nosql key-value data stores. *PVLDB*, 6(12):1434–1439, 2013.

[28] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+T: benchmarking web-scale transactional databases. In *ICDE Workshops*, pages 223–230, 2014.

[29] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*, pages 1243–1254, 2013.

[30] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.

[31] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[32] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP*, pages 54–70, 2015.

[33] Sameh Elnikety, Steven G. Dropsho, and Fernando Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, pages 117–130, 2006.

[34] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[35] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5):613–624, 2017.

[36] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[37] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD Conference*, pages 15–26, 2014.

[38] Galera Cluster. http://galeracluster.com/products/technology/, 2019.

[39] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[40] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks in a large shared data base. In Douglas S. Kerr, editor, *VLDB*, pages 428–451. ACM, 1975.

[41] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.

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

[43] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Sergio Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS*, pages 164–173, 2000.

[44] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *OSDI*, pages 185–201, 2016.

[45] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[46] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement database replication. In *VLDB*, pages 134–143, 2000.

[47] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *SIGMOD Conference*, pages 1675–1687, 2016.

[48] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *Eurosys*, pages 113–126, 2013.

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

[50] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[51] Leslie Lamport. Generalized consensus and paxos. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/, 2005.

[52] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[53] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *SOSP*, pages 104–120, 2017.

[54] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conference*, pages 21–35, 2017.

[55] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2PC transaction management in distributed database systems. In *SIGMOD Conference*, pages 1659–1674, 2016.

[56] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *NSDI*, pages 503–517, 2014.

[57] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The fuzzylog: A partially ordered shared log. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *OSDI*, pages 357–372.

[58] Yi Lu, Xiangyao Yu, and Samuel Madden. STAR: scaling transactions through asymmetric replication. *PVLDB*, 12(11):1316–1329, 2019.

[59] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB*, 7(5):329–340, 2014.

[60] Hatem A. Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9):661–672, 2013.

[61] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, pages 604–615, 2014.

[62] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196, 2012.

[63] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI*, pages 369–384, 2008.

[64] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *NSDI*, pages 453–468, 2017.

[65] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[66] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, pages 358–372, 2013.

[67] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *OSDI*, pages 479–494, 2014.

[68] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *OSDI*, pages 517–532, 2016.

[69] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *SRDS*, pages 263–273, 1999.

[70] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, pages 511–524, 2014.

[71] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *SIGMOD Conference*, pages 1279–1294, 2015.

[72] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD Conference*, pages 677–689, 2015.

[73] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, pages 305–319, 2014.

[74] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD Conference*, pages 61–72, 2012.

[75] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, pages 155–174, 2004.

[76] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. Extending DBMSs with satellite databases. *VLDB J.*, 17(4):657–682, 2008.

[77] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, pages 279–292, 2010.

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

[79] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[80] David P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, 1983.

[81] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. Design principles for scaling multi-core OLTP under high contention. In *SIGMOD Conference*, pages 1583–1598, 2016.

[82] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 6(2):145–156, 2012.

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

[84] Kun Ren, Alexander Thomson, and Daniel J. Abadi. VLL: a lock manager redesign for main memory database systems. *VLDB J.*, 24(5):681–705, 2015.

[85] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[86] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *PVLDB*, 10(4):445–456, 2016.

[87] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *SIGMOD Conference*, pages 433–448, 2019.

[88] Zechao Shang, Jeffrey Xu Yu, and Aaron J. Elmore. Rushmon: Real-time isolation anomalies monitoring. In *SIGMOD Conference*, pages 647–662, 2018.

[89] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[90] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400, 2011.

[91] 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 *VLDB*, pages 1150–1160, 2007.

[92] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and atomic rdma-based replication. In *ATC*, pages 851–863, 2018.

[93] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, 4th edition, 2014.

[94] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183, 1995.

[95] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *TODS*, 4(2):180–209, 1979.

[96] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.

[97] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12, 2012.

[98] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[99] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Samuel Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, pages 59–72, 2007.

[100] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.*, 26(4):537–562, 2017.

[101] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. *PVLDB*, 11(4):406–419, 2017.

[102] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[103] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *SIGMOD Conference*, pages 473–488, 2018.

[104] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *OSDI*, pages 233–251, 2018.

[105] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, pages 87–104, 2015.

[106] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[107] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *SIGMOD Conference*, pages 231–243, 2018.

[108] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time traveling optimistic concurrency control. In *SIGMOD Conference*, pages 1629–1642, 2016.

[109] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *PVLDB*, 11(10):1289–1302, 2018.

[110] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. BCC: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *PVLDB*, 9(6):504–515, 2016.

[111] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with RDMA networks. *PVLDB*, 12(11):1637–1650, 2019.

[112] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *SOSP*, pages 263–278, 2015.

[113] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, 2014.