Scalar DL: Scalable and Practical Byzantine Fault Detection for Transactional Database Systems

Hiroyuki Yamada
Scalar, Inc.
hiroyuki.yamada@scalar-labs.com

Jun Nemoto
Scalar, Inc.
jun.nemoto@scalar-labs.com

ABSTRACT

This paper presents Scalar DL, a Byzantine fault detection (BFD) middleware for transactional database systems. Scalar DL manages two separately administered database replicas in a database system and can detect Byzantine faults in the database system as long as either replica is honest (not faulty). Unlike previous BFD works, Scalar DL executes non-conflicting transactions in parallel while preserving a correctness guarantee. Moreover, Scalar DL is database-agnostic middleware so that it achieves the detection capability in a database system without either modifying the databases or using database-specific mechanisms. Experimental results with YCSB and TPC-C show that Scalar DL outperforms a state-of-the-art BFD system by 3.5 to 10.6 times in throughput and works effectively on multiple database implementations. We also show that Scalar DL achieves near-linear (91%) scalability when the number of nodes composing each replica increases.

PVLDB Reference Format:
Hiroyuki Yamada and Jun Nemoto. Scalar DL: Scalable and Practical Byzantine Fault Detection for Transactional Database Systems. PVLDB, 15(7): 1324-1336, 2022.
doi:10.14778/3523210.3523212

1 INTRODUCTION

Dealing with malicious attacks such as tampering with data in database systems is becoming increasingly important. The reliance of industry and government on internet services based on database systems makes such attacks more attractive and the consequences of successful attacks more critical.

Byzantine fault tolerance (BFT) techniques [8, 14, 40, 54, 70, 71] have been widely explored to tolerate such attacks. There are several extensions [28, 63, 66] of the BFT techniques to handle database transactions where multiple operations are executed in an atomic and isolated manner while exploiting the parallelism of the transactions. The BFT techniques are designed for masking Byzantine faults, and the number of replicas to mask f faulty replicas needs to be at least $3f + 1$.

Although the BFT techniques are elegant, there may be an administrative burden in practice. Specifically, when we assume malicious attacks including internal attacks, replicas need to be placed in different administrative domains[1] (ADs) [19, 44, 67] because malicious attacks are likely to be dependent, i.e., if there is one Byzantine-faulty replica in an AD, the other replicas in the same AD are also Byzantine-faulty because one fully-privileged administrator of the AD could make any malicious attacks. We could diversify replica implementations [3, 23, 27] to avoid dependent software errors or bugs, but it is not necessarily effective for malicious attacks. Therefore, a BFT system requires at least four different ADs to guarantee correctness. Managing a database system with this constraint may impose too much administrative burden or may be impractical for some organizations because most organizations have managed database systems in a single AD.

The burden can be mitigated by applying Byzantine fault detection (BFD) techniques introduced in PeerReview [32, 33]. PeerReview can only detect Byzantine faults; however, it can detect f faulty replicas with only $f+1$ replicas, i.e., it requires only two replicas (ADs) to detect one faulty replica. It could be a more practical approach for database systems when detection is acceptable.

PeerReview is a general protocol but not designed to work for databases efficiently, i.e., it cannot execute transactions in parallel while preserving a correctness guarantee (strict serializability). PeerReview could be extended to run transactions in parallel by applying concurrency control in a primary replica[2], but a secondary replica (called witness) still needs to replay the hash-chained log of the primary *sequentially* to guarantee correctness, which would limit the overall parallelism of transaction execution.

This paper presents Scalar DL, a Byzantine fault detection middleware that executes non-conflicting database transactions in parallel while preserving a correctness guarantee. Scalar DL is specifically designed for managing two database replicas that are separately administered in two ADs in a database system. We focus on using two database replicas for the following reasons: (1) Two is the lower bound for the number of replicas to deal with Byzantine faults; thus, it is the most practical setting from an administrative perspective. (2) It could be a natural extension of the current enterprise database systems with an auditor server [50] where the auditor server is securely (and separately) managed in a remote location.

Scalar DL provides a view of a single-instance database system to users and internally runs two types of database servers in separate ADs: primary database servers that manage a primary database replica holding an application's data and make all the commit decisions and secondary database servers that manage a secondary database replica holding the same data as the primary database

Proceedings of the VLDB Endowment, Vol. 15, No. 7 ISSN 2150-8097.
doi:10.14778/3523210.3523212

[1]An administrative domain (AD) is a collection of nodes and networks operated by a single organization or administrative authority.
[2]PeerReview works in a peer-to-peer manner, but we call a node that creates a record first a primary.

replica for auditing purposes. Both servers separately manage the same set of deterministic functions to derive states and results on the basis of given inputs.

The key of the Byzantine-fault detection protocol of Scalar DL is that the primary and the secondary servers make an agreement on the partial ordering of transactions in a *decentralized* and *concurrent* way. The secondary first pre-orders a transaction given from a client *partially* on the basis of conflicts (ordering phase), and the primary executes and commits the transaction that is ordered by the secondary (execution phase), and then the secondary validates the ordering result given from the primary and executes the transaction (validation phase). The three-phase protocol makes both databases derive the same correct (strict serializable) states and results as long as both ADs are honest, i.e., if either is Byzantine-faulty, their states or results would be diverged, which makes it possible for clients to observe the divergence and detect the fault in the database system.

Scalar DL is database-agnostic middleware so that it achieves the detection capability in a database system without either modifying the databases or using database-specific mechanisms. Scalar DL can currently run on PostgreSQL [31], MySQL [49], Oracle Database [53], Microsoft SQL Server [47], Apache Cassandra [24], Apache HBase [25], Amazon DynamoDB [59], Amazon Aurora [58], Azure Cosmos DB [46], and their compatible databases.[3]

Scalar DL has been used for real-world applications. The primary use case is making database records tamper-evident for digital evidence [13]. We provide solutions for regulations and laws that require tamper evidence of data. For example, regulations on data protection and privacy (e.g., GDPR and CCPA), laws of digital documents around finance and tax affairs, prior user right for intellectual property, and vehicle regulations around software updates with over-the-air (OTA) in WP.29 [22].

To the best of our knowledge, Scalar DL is the first scalable and practical approach that detects Byzantine faults in a database system that manages two database replicas separately in different ADs. This paper's contributions are as follows:

- It describes a new Byzantine fault detection protocol for a database system that manages two database replicas in different administrative domains. The detection protocol executes non-conflicting transactions in parallel (thus, achieving good scalability) while guaranteeing correctness.
- It describes the design and implementation of Scalar DL that applies the detection protocol efficiently using a middleware approach. Scalar DL is general-purpose and database-agnostic middleware so that it can be used with a wide variety of applications and database implementations.
- It provides experimental results with YCSB and TPC-C workloads to show that Scalar DL outperforms the state-of-the-art approach that extends PeerReview for database transactions when transaction concurrency can be exploited. It also presents that Scalar DL works effectively on multiple database implementations and achieves near-linear scalability when the number of nodes composing each replica increases.

The remainder of the paper is organized as follows. Section 2 describes the background and challenges. Section 3 introduces the design of Scalar DL. Section 4 describes the implementation of Scalar DL. Section 5 presents the results of our evaluation. Section 6 presents the related work. Finally, Section 7 concludes the paper.

[3]Scalar DL is built on a universal transaction manager [57] so that it can also work with non-ACID databases such as Cassandra, HBase, DynamoDB, and Cosmos DB.

2 BACKGROUND AND CHALLENGES

2.1 Background

We first describe the existing approaches that deal with Byzantine faults and discuss why they are not ideal for detecting Byzantine faults in a database system deployed to two administrative domains (ADs) environments. Note that Byzantine faults are arbitrary faults including malicious internal attacks such as tampering with data by a fully privileged administrator; thus, replicas are supposed to fail dependently in an AD [19, 44, 67], i.e., if one replica in an AD is Byzantine-faulty, the other replicas in the same AD are also Byzantine-faulty. We follow the assumption throughout the paper.

2.1.1 BFT SMR. Byzantine fault tolerance (BFT) techniques [8, 14, 40, 54, 70, 71] have been widely explored in state machine replication (SMR) to mask Byzantine behaviors. BFT SMR replicates given requests from clients using several rounds of atomic multicast between replicas to tolerate faulty replicas. The minimum number of replicas to tolerate f faulty replicas is $3f + 1$. The techniques originally execute a given input sequence serially to make a set of replicas have the same states and results. There are several extensions that make the execution more concurrent by exploiting pre-defined knowledge [36, 38].

2.1.2 BFT DB. There are several works that have extended BFT SMR to handle database transactions where multiple operations are executed in an atomic and isolated manner while exploiting the parallelism of the transactions. HRDB [66] is the first approach that applies BFT in database systems. HRDB uses a database's internal locking mechanism (strict 2PL) to make a primary replica partially order transactions and replicates the ordered transactions to secondary replicas. It requires only $2f + 1$ replicas; however, it depends on a trusted coordinator to manage which requests the coordinator can send to the secondaries concurrently. Byzantium [28] handles BFT in database transactions more generally without a trusted component using PBFT [14] (that requires $3f+1$ replicas) as a replication method. Basil [63] broadcasts each transaction to replicas who determine their votes on commit or abort independently and lets a client collect a quorum of votes. Basil uses $5f + 1$ replicas to achieve correctness with a single round-trip communication for commit.

2.1.3 BFD SMR. Byzantine fault detection (BFD) is another way of dealing with Byzantine faults. BFD only detects Byzantine faults instead of tolerating such faults; however, it requires only $f + 1$ replicas to detect f faulty replicas. PeerReview [32, 33], the state-of-the-art BFD approach, creates a total-order hash-chained execution log in a primary node and makes the other nodes (called witnesses) replay the log sequentially to compute the same states and results of the primary node and audit the primary correctness.

2.1.4 BFD DB. There are no existing BFD approaches specifically designed for a database system. Although the existing BFT DB and BFD SMR approaches could be extended to deal with BFD for database transactions, those approaches fall short in guaranteeing correctness (without a trusted component) or achieving parallel execution when deployed to a two-AD environment.

The existing BFT database systems that require $3f + 1$ replicas can be deployed to a two-AD environment by splitting the replicas into the two ADs. Suppose there are four replicas (i.e., one faulty replica can be tolerated), and each AD manages two replicas. In such a case, if one replica in an AD is faulty, the other replica in the same AD is also faulty because replicas are supposed to fail dependently in the same AD, which results in exceeding the predefined threshold for correctness. BFT systems do not give any guarantees on how they behave if the number of faulty replicas exceeds the predefined threshold and they usually have to accept the faults [19], i.e., if one of two ADs is Byzantine-faulty, BFT systems cannot even detect the fault. Although a few exceptions [16, 44] guarantee the behavior of a system when one-half replicas are faulty, they are not ideal. BFT2F [44] guarantees only fork* consistency that is weaker than linearizability; thus, it still has to accept some Byzantine faults. A2M-PBFT [16] guarantees linearizability but requires a trusted component or device, which would limit the applicability and generality.

PeerReview could be extended to execute database transactions in parallel by implementing a concurrency control mechanism. For example, PeerReview could apply two-phase locking when executing transactions in a primary. However, the witness (secondary) of the primary is required to process the total-order hash-chained log of the primary *sequentially* to guarantee correctness (i.e., strict serializability), which would limit the overall parallelism of transaction execution.

2.2 Challenges

The key challenge is to detect Byzantine faults in a database system deployed to a two-AD environment, with a correctness guarantee while exploiting the parallelism of the transactions of the database system. As further discussed in Section 3.2, (the safety side of) the correctness guarantee in this paper is that a database system provides strict serializability as long as both ADs are honest (not faulty) and can detect Byzantine faults if one of the ADs is faulty.

A strict serializability [10, 35] guarantee is required in a database system that deals with Byzantine faults. A system with (one-copy) serializability [7] only guarantees that transactions will be scheduled in an equivalent way to some serial order and does not place any constraints on what the serial order is. Thus, such a system could cause critical anomalies, so-called time-travel anomalies [1, 18], i.e., transaction T1 happens before transaction T2 in real-time, but T2 is ordered before T1 in a database system. For example, suppose there is a promotion in a bank, and you can earn bonus points if you have more than $1,000 in your bank account at the end of December. Bob originally had $1,200 in his bank account but just used $300 for some shopping, and Bob only had $900 in the bank account at the end of December. In a database system with strict serializability, Bob never earns the bonus points because Bob's account balance is not enough for it. However, if a database system only guarantees serializability, Bob might earn bonus points unexpectedly. That is because a transaction (named bonus transaction) that checks Bob's account balance and adds some bonus points to Bob's point account (that is different from the bank account) could be scheduled to be executed before Bob's $300 spending transaction even though the bonus transaction happened after the spending transaction in real-time. The time-travel anomaly is a critical issue, especially in digital evidence [13], because digital evidence is the evidence of real-time behaviors in many cases.

If PeerReview applies partial-ordering to the hash-chained log, it only guarantees serializability. That is because a primary can order conflicting transactions in a non-strict serializable manner without making the secondary notice it. Thus, PeerReview with a partially-ordered log has to accept time-travel anomalies whether they occur by chance or by a malicious activity in the primary, which means it cannot detect this type of Byzantine fault.

Dealing with Byzantine faults in a two-AD environment is also essential for a database system. Most organizations or companies manage a database system in a single AD even if the database system is a distributed database, and managing several ADs in a single organization may impose too much administrative burden or may be impractical from an administrative and operational point of view. Therefore, exploring a way to use the lowest number of ADs (i.e., two ADs) to deal with Byzantine faults is a challenge to address to make databases dealing with Byzantine faults a more practical solution.

Furthermore, the above challenges should be addressed without using trusted components from a generality and applicability perspective. It is more reasonable to assume that Byzantine faults could occur anywhere because there are usually no assumptions about the behavior of Byzantine faults. Moreover, although exploiting a trusted security device or hardware [5, 16, 42] is one of the promising ways to deal with Byzantine faults, it might not be feasible in the current cloud era because such hardware may not be widely available in the cloud.

3 SCALAR DL DESIGN

This section describes Scalar DL that effectively addresses the challenges we described in the previous section.

3.1 System Model

Scalar DL inherits the standard assumptions of prior work that deals with Byzantine faults. We assume a Byzantine fault model, i.e., Byzantine-faulty nodes behave arbitrarily, and there are no assumptions about the behavior of a fault. We assume nodes in the same AD fail dependently, but nodes in different ADs fail independently. Finally, we assume that the adversaries (and the faulty nodes they control) are computationally bound so that they cannot subvert cryptographic techniques such as a cryptographic hash function and a digital signature.

3.2 System Properties

Scalar DL provides safety and liveness if there is no fault (i.e., both ADs are not faulty). In this case, safety means that a database system provides a strict serializability [10, 35] guarantee.

If one AD is faulty, it provides safety, which means correct clients can detect a Byzantine fault in a database system if and when the Byzantine fault is observable to the clients. Therefore, if a Byzantine fault is not observable outside of a database system, even correct clients cannot detect the fault. Moreover, correct clients detect a

Byzantine fault only when the clients observe the fault from the response to a request for a database system; thus, the clients cannot detect the fault instantly.

Since it provides safety in case one AD is faulty, it cannot provide liveness in such a case due to FLP impossibility [21] unless synchrony or partial synchrony [20] is assumed. For example, even if only one AD is maliciously altered, if the AD ignores all messages and never responds, clients can never detect the fault. However, in practice, such an AD would be suspected, and the administrator of the AD would receive an inquiry; thus, the AD cannot continue ignoring messages forever.

If both ADs are faulty, Scalar DL cannot guarantee either safety or liveness. However, since we assume nodes in different ADs fail independently, it would be rare to see both ADs derive the same incorrect states and results in practice even if both ADs are faulty. Thus, correct clients might detect the faults without any guarantees.

Although Scalar DL guarantees safety even with Byzantine clients as long as either AD is honest, Scalar DL cannot prevent Byzantine clients from writing garbage data to a database system like other previous work [14, 28, 63, 66]. However, it limits the damage by providing access control; i.e., Scalar DL authenticates clients and denies access if clients do not have permissions. Similarly, Scalar DL cannot prevent Byzantine clients from returning garbage data to applications like other previous work [14, 28, 63, 66].

3.3 System Overview

3.3.1 Design Goals. The primary goal of Scalar DL is to achieve Byzantine fault detection capability in a database system while executing non-conflicting transactions in parallel.

Another important goal is to achieve the primary goal without modifying the databases or using database-specific mechanisms. We could implement the detection protocol specifically for a database implementation for better performance, but it limits the applicability. For example, we can take a similar approach to HRDB [66] to order transactions using MySQL's internal 2PL mechanism (without writing lock entries to disks); however, the approach can only work with MySQL and cannot work with PostgreSQL based on snapshot isolation.

3.3.2 System Architecture. Figure 1 shows the architecture of Scalar DL. Scalar DL consists of Scalar DL clients and Scalar DL servers. Scalar DL clients provide a view of a single-instance database system to applications. Scalar DL servers manage two database replicas (databases), a primary database replica and a secondary database replica, respectively deployed to different ADs. The servers that manage a primary database are called primary servers, and the servers that manage a secondary database are called secondary servers. Each database can be a single-node database or a multi-node distributed database that uses data partitioning and replication for high performance and crash fault tolerance.

Both Scalar DL servers maintain the same set of deterministic functions that are installed before execution. A client issues a transaction request that includes a reference to a function to execute and all the required parameters for the function. We could use SQL to create a request, but there are several challenges to be addressed. First, SQL could produce non-deterministic results. For example, ORDER BY without explicit ordering, timestamp function, and

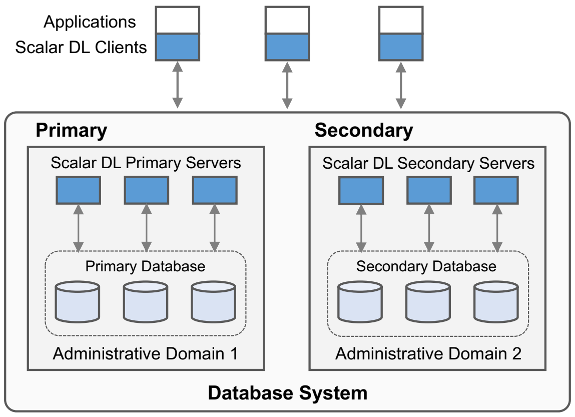


Figure 1: Scalar DL system architecture. Each database in an AD can be a single-node database or a multi-node distributed database that uses data partitioning and replication for high performance and crash fault tolerance.

auto-generation of row IDs could derive non-deterministic states or results. Thus, a system has to properly care for those, for example, by rewriting a query [66]. Second, it limits the applicability because some database implementations do not support SQL.

Scalar DL assumes a one-shot request model, which is common in OLTP systems, to decrease the likelihood of abort due to (potentially time-consuming) client-server interactions. When a request begins, it does not interact with its caller until it completes the request.

Scalar DL manages two types of secrets for message authentication codes (MACs). One is a secret shared between a client and the servers. The other is a secret shared between the primary and secondary servers. Scalar DL servers install both types of secrets before accepting clients' requests. We could use digital signatures instead of MACs[4]; however, we chose MACs for better performance.[5]

As of the current design, Scalar DL supports read, write, and delete operations but does not support predicate-based scan operations. The delete operation is a write-based logical deletion so that logically deleted records can be physically deleted as necessary at some later point.

3.3.3 Database Requirements. It is sufficient for both databases to provide ACID with strict serializability, but it is not necessary. As for the current detection protocol, it is necessary for the primary database to provide ACID with read-committed isolation and for the secondary database to provide linearizability [35] for a single operation on a single record with durability.

Scalar DL also requires both databases to be composed of a set of records where a primary key identifies each record.

3.4 The Detection Protocol

The key of the Byzantine-fault detection protocol of Scalar DL is that the primary and the secondary servers make an agreement

[4]We provide an option to use digital signatures for message authentication.
[5]We observed that digital signatures can still be three orders of magnitude slower than MACs and add non-negligible overhead to the performance of the protocol.

on the partial ordering of transactions in a *decentralized* and *concurrent* way. The protocol comprises three phases: ordering phase, commit phase, and validation phase. The secondary first pre-orders a transaction given from a client *partially* on the basis of conflicts (ordering phase), and the primary executes and commits the transaction that is ordered by the secondary (execution phase), and then the secondary validates the ordering result given from the primary and executes the transaction (validation phase). The three-phase protocol makes both databases derive the same correct (strict serializable) states and results as long as both ADs are honest, i.e., if either is Byzantine-faulty, their states or results would be diverged, which makes it possible for clients to observe the divergence and detect the fault in the database system. We explain each phase in detail.

3.4.1 Ordering Phase. The ordering phase starts when a client program accepts a transaction request from an application program for the execution of a function to the database system. A transaction request has the form $\langle n, f, a, s \rangle$, where n is a unique transaction ID that identifies the request (e.g., UUID), f is a reference to a function, a is the argument of the function, and s is a message authentication code (MAC) created for n, f, and a with a secret shared between a client and the servers. The client first sends the request to the secondary. The secondary receives the request, verifies the MAC of the request, and stores the request in the database. The request is stored with n as a primary key so that it can be used to check if the request has already been processed. If the request has already been stored, the secondary aborts the request.

Then, the secondary simulates the execution of the function to identify what records will be read and written, i.e., a read set and a write set, by the function. The simulation executes the given function without writing any records.

After the read and write set is identified, the secondary schedules the request by exploiting the read and write set using a variant of two-phase locking (2PL) [68]. Specifically, the secondary tries to create a read lock entry for the primary key of the record in the read set and create a write lock entry for the primary key of the record in the write set. A lock entry has the form $\langle k, v, c, m, h, d \rangle$, where k is the primary key of a record, v is the record version number that is incremented when the record is committed, c is a lock count that is incremented when the lock is acquired and decremented when the lock is released, m is a lock mode such as read-lock and write-lock, h is a set of lock holders (transaction IDs), and d, which is used for only write-lock entries, is a set of input dependencies expressed as a set of ⟨primary key, version number⟩ pairs that the locked entry depends on (e.g., if a function derives a record by reading ⟨k1, 2⟩ and ⟨k2, 3⟩, then the input dependencies for the record will be {⟨k1, 2⟩, ⟨k2, 3⟩}). Note that the secondary writes lock information to its database with linearizable consistency to avoid conflicting locks from being acquired and to survive crashes.

Once the secondary can take all the locks for the request, it responds to the client with a MAC created for n with the secret shared between the secondary and the primary. Otherwise, it aborts the request and releases the locks. The reason for aborting the request instead of waiting for locks to be released is that the secondary needs to re-execute the simulation to get the latest versions of records to order transactions in a strict serializable manner. However, it is not the case for read conflicts; thus, acquiring a read lock can be retried without a full restart, as discussed in Section 3.7.

The important point of the ordering phase is that it pre-orders given transaction requests on the basis of conflicts, i.e., ordering transactions partially, to exploit the parallelism of transactions while keeping the primary and the secondary always consistent. Without the ordering, conflicting requests could cause non-deterministic results, which makes the states of the primary and secondary databases diverge. We also use a variant of 2PL for transaction ordering to guarantee strict serializability in a database system [6, 10, 35]. We could use multi-version concurrency control (MVCC) for the ordering; however, it is challenging to guarantee consistent ordering between the primary and the secondary, as we discuss later in Section 3.4.4. Moreover, the 2PL-based ordering is achieved without depending on particular database-specific mechanisms.

3.4.2 Commit Phase. The commit phase starts when the client receives the acknowledgment from the secondary. The client first adds the MAC from the secondary to the transaction request and sends the updated request to the primary. The primary receives the request and verifies both MACs from the client and the secondary. If the verification fails, the primary aborts the request with a validation error.

Once the verification succeeds, the primary executes the function in the request. It uses the underlying database transaction to atomically read and write database records as specified in the function and write a transaction status (committed or aborted) with n as a key. Note that the database records are internally versioned for ordering validation described in Section 3.4.3, i.e., if a function writes a record, the function reads the latest version and creates a new version of the record.[6] Note also that the primary can schedule given requests in an arbitrary order because the requests have been already ordered in a strict serializable manner by the secondary.

Then, the primary creates a proof for each record that is written or read. Each proof has the form $\langle k, v, n, d, s \rangle$, where k is the primary key of a record, v is the record version number, n is a transaction ID, d is a set of input dependencies expressed as a set of ⟨primary key, version number⟩ pairs that the record depends on, and s is a MAC for the entries (k, v, n, and d) created with the secret shared between the primary and the secondary. The proofs show that the records have been written or read for the request and have the input dependencies (d). The primary returns the proofs and the execution result to the client.

If an error occurs during the execution, the primary aborts the transaction and returns the error with a failure status to the client. In such a case, aborted status is written as a transaction status.

3.4.3 Validation Phase. The validation phase starts when the client accepts the proofs from the primary. The client sends the proofs to the secondary before responding to the application program. The secondary receives the proofs and first verifies the MACs to see if the primary has created the proofs.

[6]Scalar DL creates a new record instead of updating the existing record to achieve traceability, but it is not necessary for the protocol to work correctly.

Once the verification succeeds, the secondary validates the proofs by comparing with the corresponding lock entries to make sure that the primary and the secondary see the same partial ordering as shown in Algorithm 1. For each proof, the secondary compares the version numbers (Line 7-18) and the input dependencies (Line 19-21) between the proof and the corresponding lock entry to check if they match. Some errors such as missing a lock entry (Line 5) or version mismatch (Line 9 and 14) could happen due to conflicting transaction recovery described in Section 3.5. Thus, it throws a potential validation error in such a case, indicating that the error could happen by a Byzantine fault or conflicting transaction recovery. This validation is pre-checking, and the final validation is always delegated to the client because the secondary could be malicious and skip the validation step.

Algorithm 1 Ordering Validation

```
 1: function VALIDATE($proofs$)
 2:     for all $p \leftarrow proofs$ do
 3:         $lock \leftarrow$ LOCKMANAGER.GET($p.key$)
 4:         if $lock$ does not exist then
 5:             throw a potential validation error
 6:         end if
 7:         if $lock.type$ = READ then
 8:             if $lock.version \neq p.version$ then
 9:                 throw a potential validation error
10:             end if
11:         else if $lock.type$ = WRITE then
12:             // the record version has been incremented
13:             if $lock.version + 1 \neq p.version$ then
14:                 throw a potential validation error
15:             end if
16:         else                                    ▷ invalid lock type
17:             throw a potential validation error
18:         end if
19:         if $lock.type$ = WRITE & $lock.inputs \neq p.inputs$ then
20:             throw a validation error
21:         end if
22:     end for
23: end function
```

If the validation succeeds, the secondary re-executes the specified function and creates database records and a result. Since each database record can be recovered lazily, as further discussed in Section 3.5, the records are not written in an atomic manner for performance reasons.[7] Once it writes all the records successfully, it increments the version number of each write lock entry acquired by the function and releases all the locks acquired by the function. Incrementing the version number of a lock and releasing the lock are done atomically. Then, it creates a set of proofs in the same way as the primary and returns the proofs and the result to the client.

The client accepts the proofs and results from both the primary and the secondary. Then the client does the final validation by comparing the values. If they are different or a validation error is thrown during the process, the client reports that there is a Byzantine fault in either the primary or the secondary AD.

3.4.4 Discussion. We discuss several design choices to derive the current protocol.

Why 2PL-based ordering? As we briefly discussed, we use a variant of 2PL for transaction ordering to guarantee strict serializability in a database system. Using multi-version concurrency control (MVCC) could schedule more transactions in parallel; however, it is challenging to guarantee consistent ordering between the secondary and the primary. Assume that conflicting transactions T1 and T2 come to the database system almost simultaneously and are scheduled together with MVCC (but cannot be scheduled together with 2PL). And assume both transactions pass the ordering phase with a serialization order T1⟶T2. However, the current protocol does not guarantee the same serialization order in the primary because the secondary does not pass explicit order dependencies such as conflict graph, which is not supposed to be feasible to manage [68], to the primary. Thus, the primary could order transactions in a different serialization order (e.g., T2⟶T1) from the secondary, which could diverge their states without the existence of Byzantine faults. In future work, we will explore another concurrency control scheme for the ordering phase to achieve better concurrency while preserving the correctness guarantee.

Why three phases? Scalar DL uses the three-phase protocol (ordering ⟶ commit ⟶ validation) to achieve parallel execution while guaranteeing strict serializability. On the other hand, PeerReview applies a two-phase protocol (which can be seen as commit ⟶ validation); however, it only guarantees serializability if it applies a parallel execution (see Section 2) since a primary can order transactions in a non-strict serializable manner. This motivates adding the ordering phase and extending the validation phase to make an agreement on strict serializable partial-ordering of transactions between the primary and the secondary in a decentralized manner.

Why starting from the secondary? The protocol could still guarantee correctness even if it starts with the primary (i.e., primary ⟶ secondary ⟶ primary); however, we chose the current flow to let the primary decide on a transaction commit.

3.5 Transaction Recovery

In the previous section, we explained the detection protocol in normal cases. However, the servers could face non-Byzantine faults such as node crashes and network failures during processing, and some locks could be left behind depending on the timing the servers face such faults.

Scalar DL recovers such left-behind locks when it reads or writes the corresponding records, as shown in Algorithm 2. When the secondary tries to acquire a lock on a record and the record is already locked, it first checks if the lock is expired (Line 2-3).[8] If the lock is not expired, it returns without recovery, and the transaction is aborted (Line 4). If it is expired, the secondary asks the primary for the statuses of the transactions by which the lock is acquired (Line 7-8). Note that the secondary not only retrieves the status of each transaction but also tries to abort the transaction to prevent the transaction from being left uncommitted or unaborted forever. If the transaction is neither committed nor aborted for some reason (e.g., the primary is too busy), it skips the recovery for the transaction

[7]We observed some performance benefits when the database is a distributed database because the cost of distributed ACID transactions is usually pretty high.

[8]We use 15 seconds for the expiration time and it is configurable.

Algorithm 2 Transaction Recovery

```
function RECOVER(key)
    lock ← LOCKMANAGER.GET(key)
    if lock is not expired then
        return                                   ▷ aborted and retried later
    end if
    // for a write lock, the number of holders is always one
    for all tid ← lock.holders do
        state ← PRIMARY.TRYABORT(tid)
        if state ≠ COMMITTED & state ≠ ABORTED then
            continue           ▷ continue to recover other transactions
        end if
        if lock.type = READ then
            LOCKMANAGER.UNLOCK(lock, tid)
        else if lock.type = WRITE then
            if state = COMMITTED then
                proof ← PRIMARY.GETPROOF(lock)
                VALIDATE(proof)                  ▷ see Algorithm 1
                newStates ← DERIVESTATES(proof)
                WRITEDATABASE(newStates, proof.key)
                LOCKMANAGER.UNLOCKANDINC(lock, tid)
            else
                LOCKMANAGER.UNLOCK(lock, tid)
            end if
        end if
    end for
end function
function DERIVESTATES(proof)
    states ← ∅
    for all input ← proof.inputs do
        record ← GETRECORD(input)          ▷ by key and version
        PUSH(states, record)
    end for
    request ← GETREQUEST(proof.tid)
    function ← GETFUNCTION(request.funcRef)
    newStates ← function(states, request.args)
    return newStates
end function
```

and checks the other transactions (Line 9-10). Skipping the recovery causes no issues because it will check (abort) the transaction when the record is accessed again. If the transaction is either committed or aborted, it recovers the lock for the transaction (Line 12-24). If the lock is read-lock, it releases the lock for the transaction (Line 12-13), i.e., removing the transaction from the lock holders and decrementing the lock count. If the lock is write-lock and the transaction is committed, the secondary asks for the corresponding proof of the lock to the primary, validates the proof as described in Algorithm 1, derives the new state of the corresponding record (Line 27-37), writes the state to the database, and releases the lock (Line 14-20). Since the new state is written for the record, it increments the version of the lock entry when unlocking it. If the transaction is aborted, it releases the lock without version increment (Line 22).

The recovery processing for each record is idempotent so that it can be retried. Even if multiple processes do the recovery simultaneously, only one process can successfully release a lock by using linearizable conditional update, i.e., a lock is released only if the lock status and the lock holders have not been changed.

3.6 Byzantine Clients

We discuss how Scalar DL handles Byzantine-faulty clients.

3.6.1 Not Following the Protocol. We first discuss a case where a Byzantine client does not follow the protocol.

First, if a Byzantine client does not send a request to the secondary before sending the request to the primary, the primary fails in verifying the request's MAC that is supposed to be created with the secret shared between the primary and the secondary. Therefore, nothing happens in the database system.

Second, if a Byzantine client does not send a request to the primary after the ordering phase, the primary never commit the request; thus, the request is treated as not executed in the database system. The secondary recovers left-behind locks eventually, as explained in Section 3.5.

Third, if a Byzantine client does not send the proofs of records created in the primary to the secondary, the secondary also recovers the situation eventually, as explained in Section 3.5.

Lastly, if a Byzantine client retransmits an old request to the servers (i.e., replay attack) after the request has already been executed, the request will not be processed in the servers because both check if the transaction ID of a request has been processed or not before writing any data. Similarly, if the proofs of created records from the primary are maliciously retransmitted to the secondary, the secondary can detect it in the validation phase, as described in Section 3.4.3. The secondary will throw an error and write no data in such a case.

In summary, even if a client does not follow the protocol, Scalar DL does not write inconsistent data in the database system.

3.6.2 Returning Tampered Results. Next, we discuss a case where a Byzantine client returns tampered data to an application. For example, a Byzantine client receives consistent results, such as Bob's account balance is \$1,000 from both the primary and the secondary, but the client returns \$500 to an application.

As discussed in Section 3.2, Scalar DL cannot avoid this like other previous work [14, 28, 63, 66] that deals with Byzantine faults because a client can return anything whether or not it receives consistent results. Even if a client is correct or can verify the results, an application can tamper with the results after it receives them.

Although Scalar DL cannot avoid this by itself, we could make applications (or even end-users) verify results if both servers add digital signatures to the results. However, it might not be feasible for some applications because it requires the whole system to keep signatures and understand how to verify results with the signatures.

Therefore, as is the case with other work, what Scalar DL guarantees is to make correct clients detect Byzantine faults in a database system. The guarantee is still acceptable for our use cases that require data to be digital evidence since a correct external auditor can obtain correct data.

3.6.3 Tampering with Data. Lastly, we discuss a case where a Byzantine client tampers with data. A Byzantine client could tamper with the requests sent from an application and the messages exchanged between the primary and the secondary through the client.

For the request sent from an application to the database system, Scalar DL cannot prevent Byzantine clients from tampering with the request like other previous work [14, 28, 63, 66] as we discussed in Section 3.2 because Scalar DL cannot identify if the request is correct or not.

For the messages (e.g., proofs) exchanged between the primary and the secondary through a client, Byzantine clients cannot successfully tamper with them because they include MACs created with the secret shared between the primary and the secondary.

3.7 Optimization

3.7.1 Spinlock for read-locking. As described in Section 3.4.1, Scalar DL aborts a transaction when it cannot acquire all required locks. However, aborting a transaction and retrying it might waste a lot of work (i.e., transaction simulation and lock acquisition) that has been done. As an optimization, Scalar DL applies spinlock for acquiring a read-lock, i.e., reading a read-lock and updating the lock are retried in a busy-loop.

This optimization cannot be naturally applied to write-locking. That is because a write-lock holder will update the holding lock entry with a new version number if it commits successfully, and a conflicting transaction that tries to acquire the write-lock will no longer see the latest lock entry without restarting the transaction.

This optimization is effective in lowering latencies, especially in a high read contention workload.

3.7.2 Parallel locking and unlocking. Since Scalar DL identifies a read set and a write set by the transaction simulation before acquiring locks, it can acquire locks in parallel. Similarly, releasing locks can also be done in parallel. Parallel locking and unlocking for a transaction do not cause deadlocks because Scalar DL aborts the transaction to re-execute the simulation if there is a conflict.

This optimization is effective in lowering latencies, especially in a low contention workload.

3.8 Correctness

In this section, we sketch a proof that our design meets the safety and liveness conditions described in Section 3.2.

3.8.1 Safety. If there is no faulty AD, it is safe because both the primary and the secondary execute the same set of transactions in the same partial order with strict serializability and always derive the same states.

If the secondary is faulty, it could arbitrarily order transactions in the ordering phase before responding to a client program, but it is still correct from a strict serializability perspective. Once the secondary partially orders a transaction and the correct primary commits the ordering in the commit phase, the secondary cannot change the ordering afterward without diverging the states from the primary. If the states diverge, correct clients can detect it. The secondary could write arbitrary states to the database or return arbitrary results to a client, but correct clients can also detect it by comparing the states and results with the ones from the primary.

If the primary is faulty, the primary could also arbitrarily order transactions. However, to proceed with the protocol successfully, the primary has to return correct ordering results to the secondary; otherwise, the correct secondary notices incorrect ordering results. The primary could buffer transactions without committing and return correct results to the secondary to deceive the secondary as if the primary successfully commits, and then the primary could reorder conflicting transactions or write arbitrary states afterward. However, it makes the primary diverge its states from the secondary, which will be detected by correct clients.

Therefore, even if one AD is faulty, it guarantees safety.

3.8.2 Liveness. If there is no faulty AD, it is live because both the primary and the secondary receive the same set of transactions in the same partial order and execute the transactions eventually.

If either the primary or the secondary is faulty, it cannot guarantee liveness because of FLP impossibility [21]; thus, it requires synchrony or partial synchrony to guarantee liveness.

4 IMPLEMENTATION

Scalar DL is mainly written in Java. Scalar DL is not a prototype and has been used in real-world applications that require data stored in a database system tamper-evident. In the current version, the primary and the secondary servers are called Ledger and Auditor, and the client program is called Client SDK. Scalar DL uses HMAC-SHA256 for message authentication codes (MACs) and ECDSA with SHA-256 hashing for digital signatures.

4.1 Ledger

Ledger implements the logic of the commit phase described in Section 3.4. Ledger also manages programmable deterministic functions called Contracts for users to create one-shot transactions. In a Contract, users can write arbitrary business logic and call database operations through the interface defined by the Contract. Nested invocation, i.e., a Contract calling another Contract, is supported so that users can implement an application's business logic with multiple Contracts modularly. Ledger executes multiple Contracts in an ACID manner by exploiting the underlying database transaction. Each Contract is stored in the database in a Java bytecode format with a digital signature attached for later verification.

Ledger abstracts the underlying database as a multi-dimensional map based on the key-value data model, which is similar to the data model of Bigtable [15]. We chose the abstraction to achieve broad applicability for various databases and data models. A record is composed of a record key (application-level primary key), a version, and a set of values, including a Contract argument used to derive the record, and a cryptographic hash of all the record values. A record key and a version form a primary key, and the primary key uniquely maps a set of values. Ledger manages the versions of records for achieving traceability. Ledger also constructs a hash-chain [39] for the records that have the same record key to make the records difficult to be maliciously altered partially, but Scalar DL does not need the hash-chain structure to provide Byzantine fault detection capability.

Ledger implements the detection protocol using the database abstraction to achieve database-agnostic property. We use Scalar DB [57], a universal transaction manager, to implement the database abstraction efficiently. The database abstraction currently supports PostgreSQL [31], MySQL [49], Oracle Database [53], Microsoft SQL

Server [47], Apache Cassandra [24], Apache HBase [25], Amazon DynamoDB [59], Amazon Aurora [58], Azure Cosmos DB [46], and their compatible databases. For those non-ACID databases such as Cassandra, HBase, DynamoDB, and Cosmos DB, Scalar DB takes care of transactions with its database-agnostic ACID transaction capability that supports snapshot isolation and strict serializability. For those ACID databases, Scalar DB provides two options: delegating transaction management to the underlying databases or doing transaction management by itself.

Ledger by only itself can provide the service to users. In such a case, Scalar DL works similarly to Oracle Blockchain Table [51] or Amazon QLDB [60] except for the database-agnostic transaction capability, i.e., it manages a database in a single AD, so it only detects some limited class of Byzantine faults.

4.2 Auditor

Auditor implements the logic of the ordering and validation phases described in Section 3.4. Auditor also manages the same Contracts as Ledger and uses the same database abstraction as Ledger so that Auditor can use various databases as the underlying database.

Auditor has to be placed in a different AD from the one where Ledger is placed to guarantee the correctness we discussed in Section 3.2.

4.3 Client SDK

Client SDK interacts with Ledger and Auditor on the basis of the protocol. An application program integrated with Client SDK manages a secret for each user and registers the secret to the databases through Ledger and Auditor respectively to be authenticated to execute Contracts. Client SDKs are written in several languages: Java, Node.js, in-browser Javascript, and Go.

Client SDK can optionally add a digital signature to a request. In such a case, the signature is stored in the record that the request creates. A record containing a digital signature can identify who has created the record, which adds extra security and traceability to the system.

5 EVALUATION

We evaluate Scalar DL to answer the following questions:

- How does Scalar DL perform compared to the state-of-the-art BFD approach (extended PeerReview)?
- How effective are Scalar DL's optimizations?
- How does Scalar DL scale as the number of database nodes increases?
- Does Scalar DL work effectively on multiple database implementations?

5.1 Benchmarked Systems

We compare Scalar DL with the state-of-the-art BFD approach, our extended version of PeerReview called PeerReviewTx. PeerReviewTx is based on the original protocol [33] and the prototype implementation [34] but extends the original protocol to work more concurrently and reasonably for transactional database systems.

PeerReviewTx applies a 2PL-based concurrency control in a primary server and executes non-conflicting transactions in parallel in the primary. Since the primary does not know what keys to lock, we attach keys to a transaction before the transaction is given to the primary, similar to the previous work that makes SMR run concurrently [36, 38]. PeerReviewTx would actually need to identify keys without such a pre-defined knowledge to make PeerReviewTx run transactions in a general way as Scalar DL does, but we chose the more efficient way for PeerReviewTx. Moreover, we implemented 2PL of PeerReviewTx using an in-memory data structure for better performance. Therefore, PeerReviewTx is a performance-enhanced version of PeerReview without almost no additional overheads. Other than the extension in the primary, PeerReviewTx works in the same way as PeerReview; the primary creates a linear hash-chained log, and a secondary server (witness) replays the hash-chained log sequentially to compute the same states of the primary. We cannot execute the secondary processing in parallel to guarantee strict serializability, as discussed in Section 2.

Since PeerReview's secondary-side auditing is challenge-based and works lazily after the primary executes a transaction, we also extended PeerReviewTx to detect faults eagerly to make it work reasonably for transactional database systems. Specifically, a client receives a log from the primary, sends the log to a secondary, and waits for the secondary to replay the log before responding to an application.

5.2 Workloads

The evaluation uses two standard workloads: YCSB and TPC-C.

YCSB [17] is a benchmark commonly used for key-value store evaluation and also adopted in transactional database evaluation by accessing multiple records in a single transaction. We used two types of workloads: Workload F (read-modify-write workload) and Workload C (read-only workload), with uniform request distribution and 100 bytes payload. For Workload F, one transaction is composed of one read-modify-write operation (one read operation and one write operation). For Workload C, one transaction is composed of two read operations. We used YCSB to clarify the basic performance of the benchmarked systems in a low contention workload.

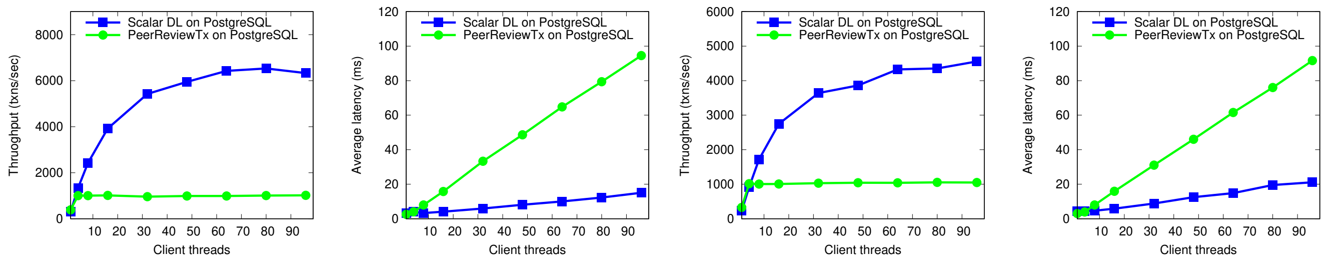
TPC-C [64] is a benchmark for online transaction processing (OLTP) databases. TPC-C has a configurable number of warehouses. We mixed two types of queries: Payment and NewOrder, with a 50/50 ratio.[9] We used TPC-C to clarify the realistic performance of the benchmarked systems in a more complex and contended workload than YCSB.

5.3 Experimental Setup

All experiments were conducted with AWS EC2 instances that run Amazon Linux 2. For each database instance, we used a c5d.4xlarge instance (8 CPU cores, 32GB memory, NVMe SSD). For clients, we used a c5.9xlarge instance (16 CPU cores, 72GB memory). We chose a big instance for clients not to make it become the bottleneck.

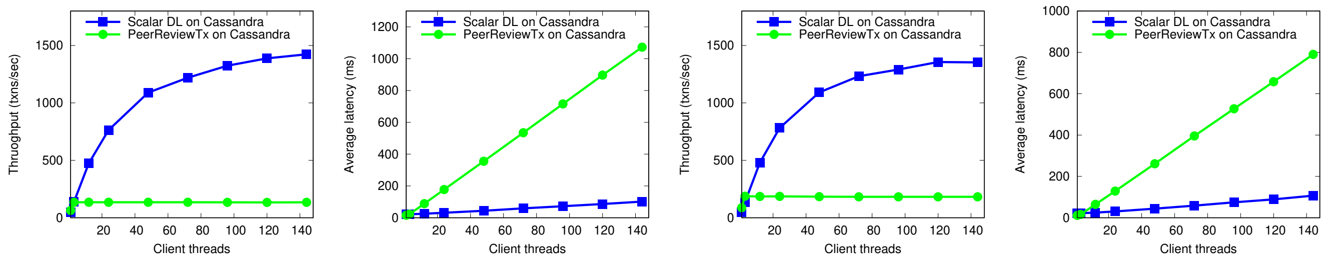
We used two types of database implementations to show the database-agnostic property of Scalar DL: PostgreSQL and Cassandra. The versions of the databases are 14.1 and 3.11.11, respectively.

[9]We chose a simpler but often-used ratio that works similarly to TPC-C full mix because NewOrder and Payment account for a large percentage (about 90%) of the full-mixed queries of TPC-C.



(a) YCSB-F (read-modify-write) throughput (b) YCSB-F (read-modify-write) latency (c) YCSB-C (read-only) throughput (d) YCSB-C (read-only) latency

Figure 2: YCSB experiments with PostgreSQL. The peak throughput of Scalar DL was about 6.5 times and 4.3 times higher than the one of PeerReviewTx in YCSB-F and YCSB-C, respectively.



(a) YCSB-F (read-modify-write) throughput (b) YCSB-F (read-modify-write) latency (c) YCSB-C (read-only) throughput (d) YCSB-C (read-only) latency

Figure 3: YCSB experiments with Cassandra. The peak throughput of Scalar DL was about 10.6 times and 7.4 times higher than the one of PeerReviewTx in YCSB-F and YCSB-C, respectively.

For PostgreSQL, we configured each instance with 500 max connections, 8GB shared buffers, 30min checkpoint timeout, 8GB max WAL size, and 512 max locks per transaction. We used a single instance PostgreSQL (that achieves linearizable single operation) with READ-COMMITTED isolation level; thus, it meets the Scalar DL requirements for databases. We configured Scalar DB to delegate transaction management to PostgreSQL.

For Cassandra, we configured each cluster with batch commitlog sync, 512 concurrent reads, and 512 concurrent writes. We used Cassandra's lightweight transactions (Paxos) to achieve linearizable operations and Scalar DB [57] to achieve ACID transactions with snapshot isolation; thus, it also meets the Scalar DL requirements for databases.

We also created two ADs, a primary AD and a secondary AD, in different networks that are assumed to be separately managed by different administrators.

We ran workloads long enough to warm up databases before experiments. We ran each experiment for 60 seconds with 10 seconds ramp-up time.

5.4 YCSB Experiments

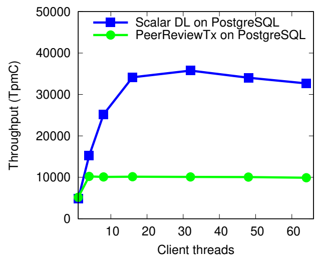
We first evaluate the throughput and latency of Scalar DL and PeerReviewTx with YCSB on the two database implementations. We loaded 100 million records before the experiments.

5.4.1 PostgreSQL. Figure 2 shows the results of both systems on PostgreSQL as we increased the number of client threads. We deployed a single PostgreSQL instance to each AD. We placed a Scalar DL Ledger and a PeerReviewTx primary server in the primary AD and a Scalar DL Auditor and a PeerReviewTx secondary server in the secondary AD.

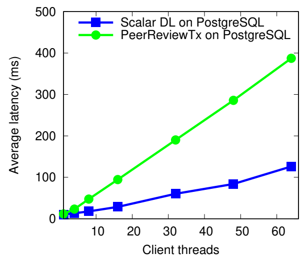
As can be seen from the results, PeerReviewTx outperformed Scalar DL when the number of client threads was small. That is because PeerReviewTx used a two-phase protocol and was more lightweight. However, when the number of client threads was more than four, Scalar DL outperformed PeerReviewTx in both throughput and latency and scaled better because it exploited the parallelism of multiple transactions. PeerReviewTx exploited some parallelism of transactions because of the concurrency control extension, but the performance improvement was saturated as it got several client concurrencies. The saturation was due to the sequential processing in the secondary server. The steeper latency increase of PeerReviewTx was also due to the sequential processing because more transactions were waiting to be processed in the secondary as the number of client threads increased.

The peak throughput of Scalar DL was about 6.5 times and 4.3 times higher than the one of PeerReviewTx in YCSB-F and YCSB-C, respectively.

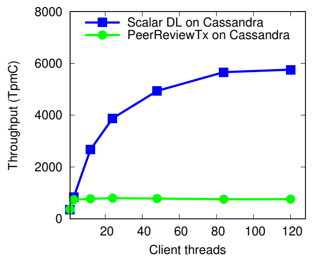
5.4.2 Cassandra. Figure 3 shows the results of both systems on Cassandra. We deployed a three-node Cassandra cluster to each AD. Each Cassandra cluster was configured to use three replicas. We placed a Scalar DL Ledger and a PeerReviewTx primary server in each node in the primary AD and a Scalar DL Auditor and a PeerReviewTx secondary server in each node in the secondary AD.



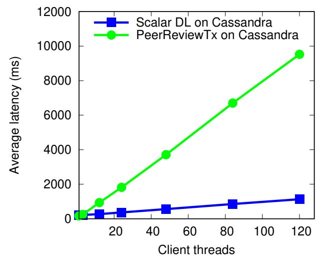
(a) Throughput with PostgreSQL.



(b) Average latency with PostgreSQL.



(c) Throughput with Cassandra



(d) Average latency with Cassandra

Figure 4: TPC-C (NewOrder and Payment mix) experiments. The peak throughputs of Scalar DL on PostgreSQL and Cassandra were about 3.5 times and 7.6 times higher than the ones of PeerReviewTx on PostgreSQL and Cassandra, respectively.

It shows similar results to the ones with PostgreSQL, but the actual performance numbers were lower. That is because Cassandra used Paxos to achieve linearizable operations, and it was a much heavier operation than a single-instance DBMS operation. Also, the universal transaction manager (Scalar DB) used to achieve ACID on Cassandra was based on the linearizable operations; thus, the Ledger-side ACID transaction was also heavier than a single-instance DBMS transaction. Since the latency of each transaction request was higher due to the heavier operations, more client threads were required until saturation. Note that each Cassandra cluster was configured with three replicas; thus, each cluster could tolerate one crash fault.

The peak throughput of Scalar DL was about 10.6 times and 7.4 times higher than the one of PeerReviewTx in YCSB-F and YCSB-C, respectively.

5.5 TPC-C Experiments

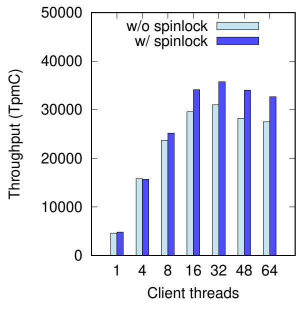
In this section, we show evaluation results with TPC-C. We loaded 100 warehouses. Figure 5 shows the throughput (TmpC) of Scalar DL and PeerReviewTx on PostgreSQL and Cassandra. We configured the systems in the same way as the previous experiments.

We can see that Scalar DL still showed performance benefits when there were more than several client threads, even with TPC-C workload. However, since TPC-C was a lot contention-heavier than the YCSB workloads, the performance improvement was less than the YCSB benchmark results. The peak throughputs of Scalar DL on PostgreSQL and Cassandra were about 3.5 times and 7.6 times higher than the ones of PeerReviewTx on PostgreSQL and Cassandra, respectively.

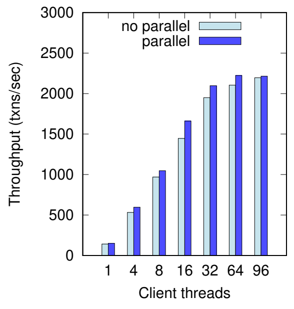
5.6 Effectiveness of Optimizations

This section evaluates the effectiveness of the two optimizations: spinlock for read-locking and parallel locking and unlocking described in Section 3.7.

Figure 5(a) shows the effectiveness of spinlock optimization in TPC-C workload. It shows that the performance improvement was greater when the number of client threads was higher. That is because spinlock for read-locking reduced the number of retries more when the concurrency was higher. We observed that the optimization improved the performance by up to 20%.



(a) Spinlock for read-locking.



(b) Parallel locking and unlocking.

Figure 5: The effectiveness of the optimizations. Spinlock and parallel locking/unlocking optimizations improved the performance by up to 20% and 15%, respectively.

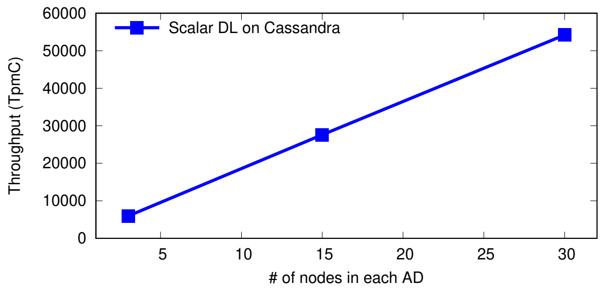


Figure 6: Scalability of Scalar DL with TPC-C. Scalar DL achieved near-linear (91%) scalability.

Figure 5(b) shows the effectiveness of parallel locking and unlocking in YCSB-F workload. We used four read-modify-write operations in a transaction. It shows that the performance improvement was greater when the number of client threads was in the middle range (around 4 to 64). That is because there were more concurrencies to exploit when there were more client threads. But, when the number of client threads was high (e.g., 96 client threads), the performance was already saturated, so increasing the parallelism did not improve the performance. We observed that the optimization improved the performance by up to 15%.

5.7 Scalability

In this section, we evaluate the scalability of Scalar DL on Cassandra with TPC-C workload. We increased the number of Cassandra nodes (instances) from 3 to 30 in each AD while keeping the number of replicas to three. We also increased the number of warehouses proportionally from 100 to 1,000.

Figure 6 shows that the throughput of Scalar DL increased near-linearly as the number of database nodes increased. Specifically, Scalar DL achieved 54,201 TpmC in a 30-node environment. Its throughput was 9.1 times higher than the one in a 3-node environment; thus, it achieved 91% scalability compared with the ideal performance (59,620 TpmC).

6 RELATED WORK

Dealing with malicious attacks for accountability in distributed systems was first introduced by Lampson [41]. SUNDR [43] is an accountable network file system for un-trusted storage servers. While it helps clients detect malicious behaviors, it is not guaranteed to detect Byzantine faults in the server-side programs. CATS [72] is another accountable network storage. It offers stronger accountability by using a trusted external publishing medium. However, it is still vulnerable to Byzantine faults, such as program tampering. PeerReview [33] is a general system providing accountability in a distributed system that consists of a collection of deterministic state machines. PeerReview detects Byzantine faults by replaying a linear hash-chained execution log using a reference implementation. However, it relies on sequential processing to guarantee correctness (i.e., strict serializability); thus, it cannot run transactions fully in parallel even if it uses a database as a state machine. Scalar DL provides a Byzantine fault detection protocol for a database system and can execute non-conflicting transactions in parallel while guaranteeing correctness.

Detecting faults in database systems has been widely explored, but most work focuses on crash faults [4, 7, 9, 29, 30, 37]. Amazon QLDB [60] and Oracle Blockchain Table [52] detect malicious behaviors in a database system using cryptography and a hash-chain data structure. However, those databases run in a single administrative domain (AD); thus, they are vulnerable to Byzantine faults. LedgerDB [69] uses a timestamp authority (TSA) as another trusted AD and puts a TSA journal on a ledger database to detect tampering after data is committed. Snodgrass [62] proposes a similar approach to LedgerDB using a trusted notarization service instead of TSA. However, they can be vulnerable to Byzantine faults such as program tampering; therefore, if a database program is maliciously altered, they cannot detect it. By contrast, Scalar DL can detect Byzantine faults in a database system without fault assumptions and trusted components as long as the faults are observable.

Byzantine fault tolerance (BFT) techniques [8, 14, 40, 54, 70, 71] have been mainly discussed in state machine replication (SMR). The techniques originally execute a given input sequence serially to make a set of replicas have the same states and results. Several works have extended BFT SMR to handle database transactions where multiple operations are executed in an atomic and isolated manner while exploiting the parallelism of the transactions. HRDB [66] is the first approach that applies BFT in database systems. HRDB uses a database's internal locking mechanism (strict 2PL) to make a primary replica partially order transactions and replicates the ordered transactions to secondary replicas. It requires only $2f+1$ replicas; however, it depends on a trusted coordinator to manage which requests the coordinator can send to the secondaries concurrently. Byzantium [28] handles BFT in database transactions more generally without a trusted component using PBFT [14] (that requires $3f+1$ replicas) as a replication method. Basil [63] broadcasts each transaction to $5f+1$ replicas who determine their votes on commit or abort independently and lets a client collect a quorum of votes. Scalar DL also executes non-conflicting transactions in parallel; however, it is designed to detect Byzantine faults in a database system that manages two separately administered database replicas.

Blockchain has appeared recently as another way of dealing with Byzantine faults. Permissionless blockchains [12, 45, 48] can be seen as extreme cases where tens of thousands of peers (e.g., personal computers and personal mobile devices that are separately administered by persons) in different ADs manage the same data to check for discrepancies between their data. The state transition of permissionless blockchains is probabilistic [56] unless a designated validator enforces transaction finality [45] because the number of ADs is unknown. Therefore, their focus is rather a special-purpose consensus based on incentive models [65]. Permissioned blockchains [2, 11, 26] have been attracting much attention, especially in financial industries. Hyperledger Fabric [2] extends BFT SMR and applies a new architecture to avoid non-deterministic operations and run transactions partially in parallel. Although permissioned blockchains have applied traditional database technologies such as transaction reordering [55, 61], they essentially share the same properties as BFT SMR and BFT databases. Thus, Scalar DL is different from those as described previously.

7 CONCLUSION

We have presented Scalar DL, a Byzantine fault detection (BFD) middleware for transactional database systems. Scalar DL executes non-conflicting transactions in parallel while preserving a correctness guarantee in a database system that manages two separately administered database replicas. Through evaluation, we have shown that Scalar DL outperformed the state-of-the-art BFD approach by 3.5 to 10.6 times in throughput and worked effectively on multiple database implementations. We have also shown that Scalar DL achieved near-linear (91%) scalability when the number of nodes composing each replica increased.

We are exploring several areas for future work: yet another concurrency control scheme for better performance and scalability, further performance optimizations without breaking database-agnostic property, SQL integration for better usability, and extending the protocol to work in separately administered three (or more) replicas environment for better availability.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Hideyuki Kawashima for their helpful comments on a draft of the paper. We are also grateful to the Scalar Engineering team, especially Toshihiro Suzuki, Yuji Ito, and Yu-Wei Su, who have made a great contribution to Scalar DB and Scalar DL.

REFERENCES

[1] Daniel Abadi. 2022. Correctness Anomalies Under Serializable Isolation. http://dbmsmusings.blogspot.com/2019/06/correctness-anomalies-under.html.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. Weed Cocco, and J. Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *EuroSys*. 1–15.

[3] A. Avizienis. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. Softw. Eng.* SE-11, 12 (1985), 1491–1501.

[4] J. F. Bartlett. 1981. A NonStop Kernel. In *SOSP*. 22–29.

[5] J. Behl, T. Distler, and R. Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *EuroSys*. 222–237.

[6] P. A. Bernstein and N. Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[7] P. A. Bernstein and N. Goodman. 1984. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Trans. Database Syst.* 9, 4 (1984), 596–615.

[8] A. Bessani, J. Sousa, and E. E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *DSN*. 355–362.

[9] A. Bhide, A. Goyal, H. Hsiao, and A. Jhingran. 1992. An Efficient Scheme for Providing High Availability. In *SIGMOD*. 236–245.

[10] Y. Breitbart, H. Garcia-Molina, and A Silberschatz. 1992. Overview of Multidatabase Transaction Management. *The VLDB Journal* 1, 2 (1992), 181–240.

[11] E. Buchman, J. Kwon, and Z. Milosevic. 2018. The Latest Gossip on BFT Consensus. *CoRR* abs/1807.04938 (2018).

[12] V. Buterin. 2013. A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/.

[13] E. Casey. 2011. *Digital Evidence and Computer Crime: Forensic Science, Computers, and the Internet* (3rd ed.).

[14] M. Castro and B. Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. 173–186.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*. 205–218.

[16] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. 2007. Attested Append-Only Memory: Making Adversaries Stick to Their Word. In *SOSP*. 189–204.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.

[18] K. Daudjee and K. Salem. 2004. Lazy Database Replication with Ordering Guarantees. In *ICDE*. 424–435.

[19] T. Distler. 2021. Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective. *ACM Comput. Surv.* 54, 1, Article 24 (2021), 38 pages.

[20] C. Dwork, N. Lynch, and L. Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (1988), 288–323.

[21] M. J. Fischer, N. A. Lynch, and M. S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382.

[22] United Nations Economic Commission for Europe. 2022. Vehicle Regulations. https://unece.org/transport/vehicle-regulations.

[23] S. Forrest, A. Somayaji, and D.H. Ackley. 1997. Building Diverse Computer Systems. In *HotOS*. 67–72.

[24] The Apache Software Foundation. 2022. Apache Cassandra. https://cassandra.apache.org/.

[25] The Apache Software Foundation. 2022. Apache HBase. https://hbase.apache.org/.

[26] The Linux Foundation. 2022. Hyperledger Sawtooth. https://sawtooth.hyperledger.org/.

[27] M. Garcia, A. Bessani, and N. Neves. 2019. Lazarus: Automatic Management of Diversity in BFT Systems. In *Middleware*. 241–254.

[28] R. Garcia, R. Rodrigues, and N. Preguiça. 2011. Efficient Middleware for Byzantine Fault Tolerant Database Replication. In *EuroSys*. 107–122.

[29] H. Garcia-Molina and B. Kogan. 1988. Achieving High Availability in Distributed Databases. *IEEE Trans. Softw. Eng.* 14, 7 (1988), 886–896.

[30] J. Gray, P. Helland, P. O'Neil, and D. Shasha. 1996. The Dangers of Replication and a Solution. In *SIGMOD*. 173–182.

[31] The PostgreSQL Global Development Group. 2022. PostgreSQL. https://www.postgresql.org/.

[32] A. Haeberlen, P. Kouznetsov, and P. Druschel. 2006. The Case for Byzantine Fault Detection. In *HotDep*.

[33] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distributed Systems. In *SOSP*. 175–188.

[34] Andreas Haeberlen, Rodrigo Rodrigues, Petr Kouznetsov, and Peter Druschel. 2022. PeerReview project homepage. http://peerreview.mpi-sws.mpg.de/.

[35] M. P. Herlihy and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[36] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *OSDI*. 237–250.

[37] B. Kemme and G. Alonso. 2000. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Trans. Database Syst.* 25, 3 (2000), 333–379.

[38] R. Kotla and M. Dahlin. 2004. High Throughput Byzantine Fault Tolerance. In *DSN*. 575–584.

[39] L. Lamport. 1981. Password Authentication with Insecure Communication. *Commun. ACM* 24, 11 (1981), 770–772.

[40] L. Lamport, R. Shostak, and M. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401.

[41] B. W. Lampson. 2004. Computer Security in the Real World. *Computer* 37, 6 (2004), 37–46.

[42] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *NSDI*.

[43] J. Li, M. Krohn, D. Mazières, and D. Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *OSDI*.

[44] J. Li and D. Maziéres. 2007. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In *NSDI*.

[45] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb. 2019. Fast and Secure Global Payments with Stellar. In *SOSP*. 80–96.

[46] Microsoft. 2022. Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/.

[47] Microsoft. 2022. SQL Server. https://www.microsoft.com/en-us/sql-server/.

[48] S. Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf.

[49] Oracle. 2022. MySQL. https://www.mysql.com/.

[50] Oracle. 2022. Oracle Audit Vault and Database Firewall. https://www.oracle.com/database/technologies/security/audit-vault-firewall.html.

[51] Oracle. 2022. Oracle Blockchain Platform Cloud Service. https://www.oracle.com/application-development/cloud-services/blockchain-platform/.

[52] Oracle. 2022. Oracle Blockchain Table. https://docs.oracle.com/en/database/oracle/oracle-database/20/newft/oracle-blockchain-table.html.

[53] Oracle. 2022. Oracle Database. https://www.oracle.com/database/.

[54] M. Pease, R. Shostak, and L. Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (1980), 228–234.

[55] P. Ruan, D. Loghin, Q. Ta, M. Zhang, G. Chen, and B. C. Ooi. 2020. A Transactional Perspective on Execute-Order-Validate Blockchains. In *SIGMOD*. 543–557.

[56] K. Saito and H. Yamada. 2016. What's So Different about Blockchain? — Blockchain is a Probabilistic State Machine. In *ICDCSW*. 168–175.

[57] Scalar. 2022. Scalar DB. https://github.com/scalar-labs/scalardb.

[58] Amazon Web Services. 2022. Amazon Aurora. https://aws.amazon.com/rds/aurora/.

[59] Amazon Web Services. 2022. Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[60] Amazon Web Services. 2022. Amazon Quantum Ledger Database (QLDB). https://aws.amazon.com/qldb/.

[61] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. 2019. Blurring the Lines between Blockchains and Database Systems: The Case of Hyperledger Fabric. In *SIGMOD*. 105–122.

[62] R. T. Snodgrass, S. S. Yao, and C. Collberg. 2004. Tamper Detection in Audit Logs. In *VLDB*. 504–515.

[63] F. Suri-Payer, M. Burke, Z. Wang, Y. Zhang, L. Alvisi, and N. Crooks. 2021. Basil: Breaking up BFT with ACID (Transactions). In *SOSP*. 1–17.

[64] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C (Revision 5.11. http://www.tpc.org/tpcc/.

[65] F. Tschorsch and B. Scheuermann. 2016. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys Tutorials* 18, 3 (2016), 2084–2123.

[66] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. 2007. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *SOSP*. 59–72.

[67] M. Vukolić. 2010. The Byzantine Empire in the Intercloud. *SIGACT News* 41, 3 (2010), 105–111.

[68] G. Weikum and G. Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers Inc.

[69] X. Yang, Y. Zhang, S. Wang, B. Yu, F. Li, Y. Li, and W. Yan. 2020. LedgerDB: A Centralized Ledger Database for Universal Audit and Verification. *PVLDB* 13, 12 (2020), 3138–3151.

[70] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*. 253–267.

[71] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC*. 347–356.

[72] Aydan R. Yumerefendi and Jeffrey S. Chase. 2007. Strong Accountability for Network Storage. In *FAST*.