Noname manuscript No.
(will be inserted by the editor)

ByShard: Sharding in a Byzantine Environment

Jelle Hellings · **Mohammad Sadoghi**

Received: date / Accepted: date

Abstract The emergence of blockchains has fueled the development of resilient systems that deal with *Byzantine failures* due to crashes, bugs, or even malicious behavior. Recently, we have also seen the exploration of *sharding* in these resilient systems, this to provide the scalability required by very large data-based applications. Unfortunately, current sharded resilient systems all use system-specific specialized approaches toward sharding that do not provide the flexibility of traditional sharded data management systems. To improve on this situation, we fundamentally look at the design of sharded resilient systems. We do so by introducing ByShard, a unifying framework for the study of sharded resilient systems. Within this framework, we show how *two-phase commit* and *two-phase locking*—two techniques central to providing *atomicity* and *isolation* in traditional sharded databases—can be implemented efficiently in a Byzantine environment, this with a minimal usage of costly Byzantine resilient primitives. Based on these techniques, we propose *eighteen* multi-shard transaction processing protocols. Finally, we practically evaluate these protocols and show that each protocol supports high transaction throughput and provides scalability while each striking its own trade-off between *throughput*, *isolation level*, *latency*, and *abort rate*. As such, our work provides a strong foundation for the development of ACID-compliant general-purpose and flexible sharded resilient data management systems.

Keywords Sharding · Resilient System · Byzantine Fault-Tolerance · Two-Phase Commit · Two-Phase Locking

An extended abstract of this work appeared at the 47$^{\text{th}}$ International Conference on Very Large Data Bases (VLDB 2021) [26].

Jelle Hellings
Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, ON, Canada.
E-mail: jhellings@mcmaster.ca

Mohammad Sadoghi
Exploratory Systems Lab, Department of Computer Science, University of California, Davis, One Shields Avenue, Davis, CA, USA.
E-mail: msadoghi@ucdavis.edu

1 Introduction

The emergence of blockchains is fueling interest in new *resilient systems* that provide data and transaction processing in the presence of *Byzantine behavior*, e.g., faulty behavior originating from software, hardware, or network failures, or from coordinated malicious attacks [19, 21, 2, 49, 15, 41, 20]. These blockchain-inspired systems are attractive, as they can provide resilience among many independent participants [19, 40, 31]. Due to these qualities, interest in blockchains is widespread and includes applications in health care, IoT, finance, agriculture, and the governance of supply chains for fraud-prone commodities (e.g., such as hardwood and fish) [34, 17, 38, 53, 45, 35, 50]. As such, blockchain-inspired systems can *prevent service disruption* due to failures that compromise part of the system and can *improve data quality* of data that is managed by many independent parties, potentially reducing the huge costs associated with both [8, 33].

Unfortunately, typical blockchain-inspired systems utilize a fully-replicated design in which every participating replica holds all data and processes all transactions, which is at odds with the scalability requirements of modern very large data-based applications [43, 44]. Consequently, recent blockchain-inspired data processing systems such as AHL [11], Caper [2], Cerberus [25], Chainspace [1], and SharPer [3] propose to provide *scalability* by introducing *sharding*: instead

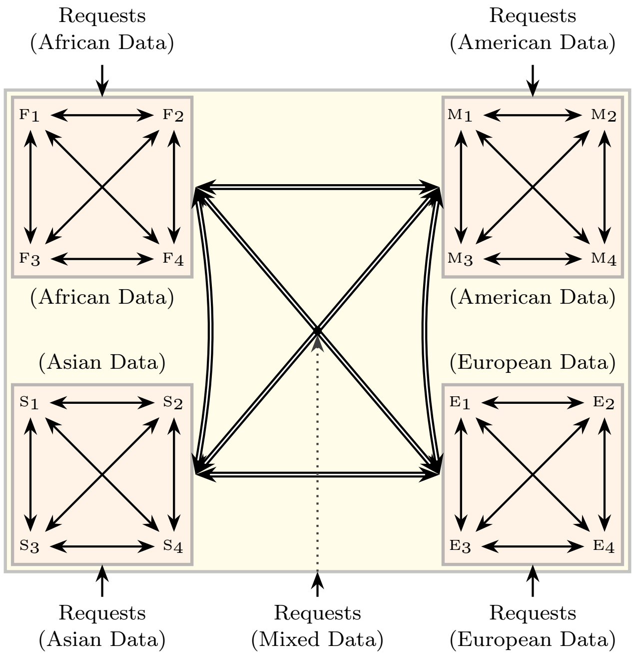

Figure 1 A *geo-scale aware sharded* design in which four resilient clusters hold only a part of the data. Local decisions within a cluster are made via *consensus* (normal arrows), whereas multi-shard coordination to process multi-shard transactions requires *cluster-sending* (double-lined arrows).

of operating a single fully-replicated system, one partitions the data (e.g., based on location) among several independently-run blockchain-based resilient clusters that each operate as a single shard using consensus and communicate with each other via cluster-sending. We have sketched this design in Figure 1.

In such a sharded design, several resilient clusters together maintain all data, while each cluster only holds part of the data. Consequently, sharded designs provide *storage scalability* as adding shards increases overall storage capacity. Furthermore, sharded designs promise *processing scalability* as transactions on data held by different shards can be processed in *parallel*. To deliver on the promises of sharding, one needs an efficient way to process *multi-shard transactions* that affect data on multiple shards, however [42].

Unfortunately, existing sharded resilient systems use system-specific solutions to provide multi-shard transaction processing: they either are mainly optimized for single-shard transactions [11], optimized for transactions that do not content for the same resources [2,3], or depend on the specifics of a UTXO-based data model to deal with contention [1]. This is in contrast with traditional distributed databases that provide application-agnostic ACID-compliant data and transaction processing that is tunable to a wide range of application-specific requirements. For example, by offering flexible multi-shard transaction processing using two-phase commit [18, 42,46] and two-phase locking [42].

This raises the question whether such flexible multi-shard transaction capabilities can be provided in a Byzantine environment. In this paper, we positively answer this question in three steps. First, we take a structured look at providing resilience in a Byzantine environment and how this affects sharded transaction processing. Next, we introduce the ByShard framework, a formalization of sharded resilient systems, and show how the design principles of traditional distributed databases can be expressed within this framework. Finally, we use the ByShard framework to evaluate the resulting design space for multi-shard transaction processing in a Byzantine environment.

To process multi-shard transactions, ByShard introduces the *orchestrate-execute model* (OEM). This model can incorporate all commit, locking, and execution operations required for processing a multi-shard transaction in at-most two consensus steps per involved shard. The first component of OEM is *orchestration*: the replication of transactions among all involved shards while also *reaching an atomic decision* on whether the transaction can be committed or not. To provide orchestration, we show how to adapt *two-phase commit style* orchestration to a Byzantine environment at a minimal cost (in terms of consensus steps at the involved shards). In specific:

1. We provide *linear orchestration* that minimizes the overall number of consensus and cluster-sending steps necessary to reach an agreement decision, this at the cost of latency.
2. We provide *centralized orchestration and distributed orchestration* that both minimize the latency necessary to reach an agreement decision by reaching such decisions in at-most three or four consecutive consensus steps, respectively, this at the cost of additional consensus and cluster-sending steps.
3. To enable centralized and distributed orchestration, we introduce Byzantine primitives to process *all* commit and abort votes using only a single consensus step per involved shard.

The second component of OEM is *execution* of transactions. To provide execution capabilities that maintain *data consistency* among shards, we show how to adapt standard *two-phase locking style* execution to a Byzantine environment at a minimal cost (in terms of consensus steps at the involved shards). In specific:

4. We introduce Byzantine primitives to provide *blocking locks* that can be processed without any additional consensus steps for the involved shards. Fur-

thermore, we show how these primitives also support *non-blocking locks*.

5. Based on these primitives, we show how *read uncommitted*, *read committed*, and *serializable* execution of transactions can be provided.
6. As a baseline for comparison, we also include isolation-free execution.

These orchestration and execution methods result in *eighteen* practical protocols for processing multi-shard transaction. To further showcase the flexibility of ByShard, we show that both AHL [11] and a generalization of Chainspace [1] can be expressed within OEM. We refer to Table 1 for an analytical comparison of each of these protocols. Finally, we apply the above techniques to a data and transaction model representative for a Byzantine sharded environment and evaluate the behavior of the resulting designs in *eight* experiments:

7. Our evaluation shows that all eighteen ByShard protocols can effectively deal with multi-shard transaction workloads and have excellent *scalability*: increasing the number of shards will always decrease the work done per shard.
8. Furthermore, all eighteen ByShard protocols have excellent transaction *throughput* when contention is low. When contention is high, the protocols each make their own trade-off between *isolation level*, *latency*, and *abort rate* while maximizing throughput.

As such, we believe that our work provides a solid foundation for the development of flexible *general-purpose* scalable Byzantine data management systems.

2 Background on Resilience

Before we look at the design of sharded resilient systems, we take a look at the operations of traditional (non-sharded) resilient systems that can deal with *Byzantine* behavior (e.g., replicas that crash, behave faulty, or act malicious). Typical resilient systems process a transaction τ requested by client c by performing five steps:

1. first, τ needs to be *received* by the system;
2. second, τ must be reliably *replicated* among all replicas in the system;
3. third, the replicas need to agree on an *execution order* for τ;
4. next, the replicas each need to *execute* τ and *update* their current state accordingly; and
5. finally, client c needs to be *informed* about the result.

At the core of resilient systems are *consensus protocols* [7,9,19,37,38] that coordinate the operations of individual replicas in the system by *replicating* transactions among all non-faulty replicas in a fault-tolerant manner, e.g., a Byzantine fault-tolerant system driven by Pbft [9] or a crash fault-tolerant system driven by Paxos [37]:

Definition 1 A *consensus protocol* coordinates decision making among the replicas of a resilient cluster (of replicas) $\mathcal{S}$ by providing a reliable ordered replication of *decisions*. To do so, consensus protocols provide the following guarantees:.[1]

1. if non-faulty replica $\textsc{r} \in \mathcal{S}$ makes an i-th decision, then all non-faulty replicas $\textsc{r}' \in \mathcal{S}$ will make an i-th decision (whenever communication becomes reliable);
2. if non-faulty replicas $\textsc{r}_1, \textsc{r}_2 \in \mathcal{S}$ make i-th decisions D_1 and D_2, respectively, then $D_1 = D_2$ (they make the same i-th decisions); and
3. whenever a non-faulty replica learns that a decision D needs to be made, then it can force a consensus decision on D.

Resilient systems operate in rounds, and in each round consensus is used to decide on and replicate a single transaction (or a set of transactions if batching is used [19]). The round in which a transaction is replicated also determines a linearizable *execution order*. Hence, replication of a transaction and agreeing on an execution order (steps 2 and 3 above) are *a single consensus step*. In practical deployments of resilient systems, reaching consensus on a decision is costly and takes a rather long time. We illustrate this next.

Example 1 Consider a deployment of the Pbft consensus protocol [9,19,21]. To maximize resilience and to deal with disruptions at any location, individual replicas need to be spread out over a wide-area network, e.g., spread-out in North America. Due to the spread-out nature of the system, the message delay between replicas is high, and a message delay of $\delta = 10\,\text{ms}$ is at the low end [11,20].

Pbft operates via a *primary-backup* design in which, under normal conditions, a designated replica (the primary) is responsible for proposing decisions to all other replicas (the backups). The primary does so via a PrePrepare message. Next, all replicas exchange their local state to determine whether the primary properly proposed a decision. To do so, all replicas participate

[1] We provide a *practical* definition of consensus. In practice, decisions will be made on *external requests* (guaranteeing non-triviality) if such requests are available to non-faulty replicas (guaranteeing *termination*). Theoretical definitions typically have more abstract requirements for *termination* and *non-triviality*.

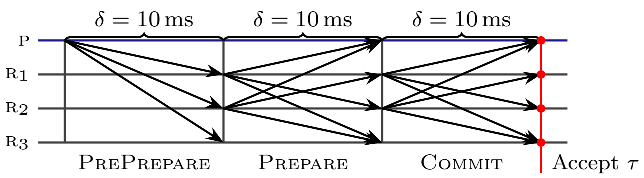

Figure 2 A schematic representation of the normal-case of Pbft: the primary P proposes transaction τ to all replicas via a PrePrepare message. Next, replicas commit to τ via a two-phase all-to-all message exchange. In this example, replica R3 is faulty and does not participate.

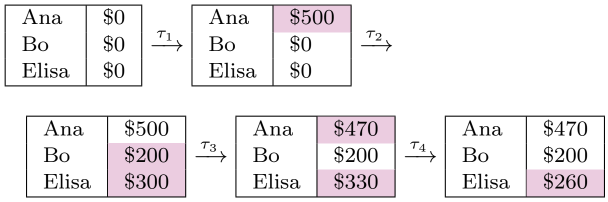

Figure 3 Evolution of the *current state* while executing the transactions of Example 2.

in two phases of all-to-all communication (via Prepare and Commit messages). Hence, if the message delay is δ, then it will take at least 3δ (first the PrePrepare phase, then the Prepare phase, and, finally, the Commit phase) before a proposed decision is accepted by all replicas: e.g., with $\delta = 10\,\mathrm{ms}$, it will take at least $3\delta = 30\,\mathrm{ms}$ for Pbft to decide on a transaction after the primary received that transaction. In Figure 2, we have illustrated this basic working of Pbft.

In a naive implementation of Pbft, the message delay ultimately limits the transaction throughput: if the $(\rho+1)$-th consensus decision will be made *sequentially after* the ρ-th decision, then the resulting throughput will be at-most $1/(3\delta) \approx 33\,\mathrm{txn/s}$ in the sketched environment. To increase performance, Pbft implementations can use *out-of-order processing* in which replicas can work on several consensus rounds at the same time [9,11,21,19]. If, for example, individual replicas have sufficient network bandwidth and memory buffers available, then a fine-tuned out-of-order Pbft can easily reach 1000 txn/s. Furthermore, batching can be used such that each consensus decision itself represents many transactions, resulting in systems that can reach even higher throughputs. The high cost of consensus is not specific to Pbft and is shared by all other popular consensus protocols, e.g., in HotStuff [55], each consensus decision will take at least $7\delta = 70\,\mathrm{ms}$ in the sketched environment.

To assure that all non-faulty replicas have *the same state*, transactions are executed in the linearizable order determined via consensus and must be *deterministic* in the sense that execution must always produce exactly the same results given identical inputs:

Example 2 Consider a banking system in which each transaction changes the balance of one or more accounts. The *current state* is the balance of each account and can be obtained from the initial state by executing each transaction in-order. Consider the first four transactions

τ_1 = "add \$500 to *Ana*";

τ_2 = "add \$200 to *Bo* and \$300 to *Elisa*";

τ_3 = "move \$30 from *Ana* to *Elisa*";

τ_4 = "remove \$70 from *Elisa*"

(in which the balance of each account is referred to by the name of the account holder). After execution of these transactions, the current state evolves as illustrated in Figure 3.

As all replicas maintain exactly the same (fully-replicated) state and, using consensus, replicate exactly the same transactions and determine exactly the same execution order, each replica can *execute* each transaction and *update* their current state fully independent (without any further need to exchange information). Hence, in a resilient system, transaction processing can be reduced to the *single* problem of ordered transaction replication, which is solved by off-the-shelf consensus protocols [9,37,55] (independent of the data and transaction model supported by the system).

Here, we assume that transactions are always replicated and executed as a whole. To deal with transactions that are *not applicable*, e.g., that violate constraints, we can include *abort* as a legitimate execution outcome (that does not affect the current state). This assumption is essential to reliably deal with Byzantine behavior: all decisions—including the decision that a transaction is not applicable—need to be made by *all non-faulty replicas* (via consensus), this to ensure that Byzantine replicas cannot force such a decision or interfere with reliably making such a decision.

Example 3 Consider the banking system of Example 2. After execution of τ_1, τ_2, τ_3, and τ_4, Ana has a balance of \$470. Now consider transaction τ_5 = "move \$500 from *Ana* to *Bo*". If the system prevents negative account balances, then τ_5 cannot be successfully executed after τ_4. Hence, if τ_5 is replicated and scheduled for execution right after τ_4, then the transaction must be *aborted* at all replicas, and the client needs to be informed of this abort.

3 Sharded resilient systems

In the previous section, we detailed the operations of traditional non-sharded resilient systems: we outlined *five* steps resilient systems perform to process transactions in a *Byzantine environment* and showed that all necessary coordination and communication between replicas in such a system is restricted to a single ordered replication step, which is handled via consensus.

The step from a non-sharded to a sharded resilient system complicates the processing of transactions significantly. To illustrate this, we revisit the five steps for processing a transaction in a resilient system. Consider a multi-shard transaction τ processed by a resilient system and assume we know which shards are involved in processing τ. First, the transaction τ needs to be replicated to all replicas of all shards involved in executing τ. After this, the replicas need to agree an *execution order* for τ. In fully-replicated systems both steps are solved at once using system-wide consensus, as the replication order determines a linearizable execution order. In a sharded system, per-shard replication of τ only yields a local linearizable replication order within that shard, however: as distinct shards can replicate transactions locally in different orders, the local replication order does not necessary determine a conflict-free execution order for τ across shards (e.g., serializable execution [4,5,22]). Hence, determining an execution order of τ across shards—necessary to maintain data consistency across shards—requires further coordination between the involved shards.

Besides determining the execution order, also *execution* and *updating* the state of replicas poses a challenge in a sharded environment. Within traditional systems, individual replicas can independently execute transactions and update their state accordingly, as each replica holds a full copy of all data. This no longer holds for multi-shard transaction: each replica only holds a copy of the data in its shard. Hence, for the execution of τ, replicas in the involved shards need to exchange any necessary state. This exchange is complicated by the presence of Byzantine replicas in each of the involved shards and, hence, requires additional coordination to assure that all necessary state is reliably exchanged.

Next, we will step-wise address these challenges towards multi-shard transaction processing in sharded resilient systems. First, we introduce the ByShard framework, a formalization of sharded resilient systems. Next, we present the *orchestrate-execute model* (OEM) used by ByShard to process multi-shard transactions. Then, in Section 4, we propose orchestration methods inspired by two-phase commit. Next, in Section 5, we propose execution methods inspired by two-phase locking. Finally, in Section 6, we evaluate the performance of transaction processing via OEM in ByShard.

3.1 A resilient sharding framework

Let $\mathcal{R}$ be a set of replicas. We model a *sharded system* as a partitioning of $\mathcal{R}$ into a set of $\mathbf{z}$ shards $\mathfrak{S} = \{\mathcal{S}_1, \dots, \mathcal{S}_{\mathbf{z}}\}$. Let $\mathcal{S} \in \mathfrak{S}$ be a shard. We write $\mathbf{n}_{\mathcal{S}} = |\mathcal{S}|$ to denote the number of replicas in $\mathcal{S}$ and $\mathbf{f}_{\mathcal{S}} = |\mathcal{S}|$ to denote the Byzantine faulty replicas in $\mathcal{S}$. We assume $\mathbf{n}_{\mathcal{S}} > 3\mathbf{f}_{\mathcal{S}}$, a minimal requirement to deal with Byzantine behavior within a single shard in practical settings [13,14]. Let τ be a transaction. We write $\text{shards}(\tau) \subseteq \mathfrak{S}$ to denote the shards that are affected by τ (the shards that contain data that τ reads or writes). We say that τ is a *single-shard transaction* if $|\text{shards}(\tau)| = 1$ and a *multi-shard transaction* otherwise.

Example 4 Consider a banking system similar to that of Example 2. This time, however, the system is *sharded* into twenty-six shards $\mathfrak{S} = \{\mathcal{S}_a, \dots, \mathcal{S}_z\}$, one for each letter of the alphabet, such that the shard $\mathcal{S}_\alpha$, $\alpha \in \{a, \dots, z\}$, holds accounts of people whose name starts with α. Now reconsider the transactions of Example 2. We have $\text{shards}(\tau_1) = \{\mathcal{S}_a\}$, $\text{shards}(\tau_2) = \{\mathcal{S}_b, \mathcal{S}_e\}$, $\text{shards}(\tau_3) = \{\mathcal{S}_a, \mathcal{S}_e\}$, and $\text{shards}(\tau_4) = \{\mathcal{S}_e\}$. Hence, transactions τ_1 and τ_4 are single-shard transactions, whereas τ_2 and τ_3 are multi-shard transactions.

Within ByShard, we can employ any *consensus protocol* [7,9,37,38] to make decisions within a shard, which allows us to operate shards as if they are a single-replica shard. We assume that consensus protocols in ByShard only make *valid* decisions: each decision made by a shard $\mathcal{S}$ will reflect a single processing step at that shard of some transaction. Within ByShard, shards perform consensus independently of each other. Hence, different shards can concurrently make distinct consensus decisions. We also need a Byzantine resilient primitive that enables coordination *between* shards. For this role, we can choose any *cluster-sending protocol* [28,27] that provides reliable communication between shards:

Definition 2 A *cluster-sending protocol* provides reliable communication between resilient clusters $\mathcal{S}_1$ and $\mathcal{S}_2$. To enable $\mathcal{S}_1$ to send a value v to $\mathcal{S}_2$, cluster sending protocols provide the following guarantees:

1. $\mathcal{S}_1$ is able to send v to $\mathcal{S}_2$ only if there is *agreement* on sending v among the non-faulty replicas in $\mathcal{S}_1$;
2. all non-faulty replicas in $\mathcal{S}_2$ will *receive* the value v; and
3. all non-faulty replicas in $\mathcal{S}_1$ obtain *confirmation* of receipt.

In ByShard, cluster-sending steps always follow consensus decision. Hence, agreement on any cluster-sending step will be reached without further consensus overhead.

3.2 The orchestrate-execute model

Consider a multi-shard transaction τ. To process this transaction, we will require *commit steps* to replicate the transaction among all replicas in all involved shards and to reach an atomic decision on whether to commit or abort τ. Furthermore, we will require *locking steps* to provide isolated execution, guaranteeing a consistent execution order among all shards, and *execution steps* that update the state of individual replicas.

At the same time, we want to *minimize* the number of consensus decisions at each involved shard to implement these commit, locking, and execution steps. To do so, we propose the *orchestrate-execute model* (OEM) that is able to incorporate the necessary commit, locking, and execution steps required for processing a multi-shard transaction in at-most two consensus steps per involved shard. In OEM, processing of a multi-shard transaction τ is modeled via individual *shard-steps* that are performed independently by each shard in $\text{shards}(\tau)$ via consensus. Each shard-step of $\mathcal{S} \in \text{shards}(\tau)$ can inspect local data at $\mathcal{S}$, modify local data at $\mathcal{S}$, and forward execution to other shards via cluster-sending:

Example 5 Consider the sharded banking example of Example 4 and consider the transaction

$\tau =$ "if *Ana* has \$500 and *Bo* has \$200, then
move \$400 from *Ana* to *Elisa*;
move \$100 from *Bo* to *Elisa*",

requested by client c. We have $\text{shards}(\tau) = \{\mathcal{S}_a, \mathcal{S}_b, \mathcal{S}_e\}$. Next, we rewrite τ to a processing plan with a minimal number of shard-steps (on success). This plan has four shard-steps, namely:

$\sigma_1 =$ "if *Ana* has \$500,
then remove \$400 from *Ana*; $\Longrightarrow_{\mathcal{S}_b}(\sigma_2)$
else send `failure` to c"
$\sigma_2 =$ "if *Bo* has \$200,
then remove \$100 from *Bo*; $\Longrightarrow_{\mathcal{S}_e}(\sigma_3)$
else $\Longrightarrow_{\mathcal{S}_a}(\sigma_4)$"
$\sigma_3 =$ "add \$500 to *Elisa* and send `success` to c"
$\sigma_4 =$ "add \$400 to *Ana* and send `failure` to c"

In which $\Longrightarrow_{\mathcal{S}}(\sigma)$ represents a cluster-sending step that forwards execution to shard $\mathcal{S}$, which is then instructed to execute shard-step σ. For simplicity, we omitted any locking from this processing plan. Hence, this plan results in a non-isolated execution that can violate *consistency constraints* on the data. Notice that the shards affected by processing τ depend on the *current state*: depending on the current state of $\mathcal{S}_a$ and $\mathcal{S}_b$, either only $\mathcal{S}_a$ is affected, or $\mathcal{S}_a$ and $\mathcal{S}_b$ are affected, or $\mathcal{S}_a$, $\mathcal{S}_b$, and $\mathcal{S}_e$ are affected.

OEM overlaps the operations necessary for providing *atomicity*, *isolation*, and *consistency* [4,5,22] to minimize the number of consensus steps. For this design, OEM utilizes only *three types* of shard-steps per shard:

Vote-step A *vote-step* $\text{Vote}(\mathcal{S})$ for $\mathcal{S}$ verifies constraints to determine whether $\mathcal{S}$ votes to either *commit* or *abort*. Furthermore, the vote-step can make local changes, e.g., modify local data or acquire locks. To simplify presentation, we assume that a vote-step yielding an *abort* vote does not have any side-effects.

Commit-step A *commit-step* $\text{Commit}(\mathcal{S})$ for $\mathcal{S}$ performs necessary operations to finalize τ when τ is committed, e.g., modify data and release locks obtained during a preceding vote-step.

Abort-step An *abort-step* $\text{Abort}(\mathcal{S})$ for $\mathcal{S}$ performs necessary operations to roll back τ when τ is aborted, e.g., roll back local changes of a preceding vote-step or release locks obtained during a preceding vote-step.

Whether a vote-step, commit-step, or abort step is necessary for a given shard $\mathcal{S}$ when processing a transaction τ with $\mathcal{S} \in \text{shards}(\tau)$ depends on the details of τ and on the execution method used (see Section 5):

Example 6 Consider the processing plan for τ of Example 5. The shard-steps σ_1 and σ_2 are *vote-steps* that decide whether τ can commit by checking the balance of Ana and Bo. The shard-step σ_3 is a *commit-step* that finalizes execution. Finally, shard-step σ_4 is an *abort-step* that rolls back the modifications made by vote-step σ_1. This abort-step is only executed if *Ana* has \$500 (hence, σ_1 removed \$400), but *Bo* does not have \$200. Note that there is *no* abort-step for shards $\mathcal{S}_b$ and $\mathcal{S}_e$, as no changes are made to accounts on these shard *before* a commit decision was made (by σ_2).

In the following two sections, we will discuss how to process multi-shard transactions using these three shard-steps with minimal cost (in terms of consensus and cluster-sending steps).

4 Providing orchestration

Let τ be a multi-shard transaction. The first part of processing τ is to orchestrate the replication of τ to the involved shards in shards(τ), assure that all these shards reach an *atomic* decision on whether to commit (and execute τ) or to abort (and cancel execution of τ), and trigger the corresponding commit-steps or abort-steps. As such, orchestration mimics the role of *commit protocols* in traditional sharded data management systems [18,42,46]. Next, we introduce the three orchestration methods of ByShard.

4.1 Linear orchestration

First, we propose an orchestration method based on the traditional *linear two-phase commit protocol* (Linear-2PC) [18,42].

Let $\mathcal{S}_1, \dots, \mathcal{S}_n$ be an ordering of all shards $\mathcal{S}_1, \dots, \mathcal{S}_n \in \text{shards}(\tau)$ with vote-steps. The transaction is orchestrated towards a decision by starting execution of $\text{Vote}(\mathcal{S}_1)$. If execution of $\text{Vote}(\mathcal{S}_i)$, $1 \leq i < n$, results in a *commit* vote, then $\mathcal{S}_i$ forwards execution of τ to $\mathcal{S}_{i+1}$, after which $\mathcal{S}_{i+1}$ will start execution of $\text{Vote}(\mathcal{S}_{i+1})$. If execution of $\text{Vote}(\mathcal{S}_n)$ results in a *commit* vote, then τ will be committed. To do so, $\mathcal{S}_n$ forwards execution of τ to all shards $\mathcal{S} \in \text{shards}(\tau)$ with a commit-step $\text{Commit}(\mathcal{S})$, after which each such shard will execute $\text{Commit}(\mathcal{S})$ *in parallel*. Finally, if execution of $\text{Vote}(\mathcal{S}_i)$, $1 \leq i \leq n$, results in an *abort* vote, then τ will immediately be aborted without further vote-steps (*fast-abort*). To do so, $\mathcal{S}_i$ forwards execution of τ to all shards $\mathcal{S} \in \{\mathcal{S}_1, \dots, \mathcal{S}_{i-1}\}$ with an abort-step $\text{Abort}(\mathcal{S})$, after which each such shard will execute $\text{Abort}(\mathcal{S})$ *in parallel*. We illustrated linear orchestration in Figure 4, *left*.

Theorem 1 *Let τ be a transaction with n_v vote-steps, n_c commit-steps, and n_a abort-step. Using* linear orchestration*, τ can be committed (aborted) in $n_v + 1$ (in at-most $n_v + 1$) consecutive consensus steps using $n_v + n_c$ (using at-most $n_v + n_a$) consensus steps and using $n_v + n_c - 1$ (using at-most $n_v + n_a - 1$) cluster-sending steps.*

Proof Assume that τ is committed. In this case, the n_v vote-steps are performed in sequence, after which all n_c commit-steps are performed in parallel. Hence, we use $n_v + n_c$ consensus steps, of which $n_v + 1$ need to be consecutive. To forward execution, $n_v + n_c - 1$ cluster-sending steps are performed. The case in which τ is aborted is analogous.

The main strengths of linear orchestration are its simplicity, the flexibility in the order in which vote-steps are processed, and its ability to *abort-fast*. As linear orchestration will only perform abort-steps at previously-voted shards, one can minimize the number of abort-steps by first processing vote-steps of shards with *only vote-steps*, and only after that the shards with both vote- and abort-steps. Furthermore, if heuristics are available, then linear orchestration can prioritize vote-steps with high likelihood of constraint failure in an attempt to quickly arrive at an abort decision. Finally, we can eliminate the commit-step or abort-step for $\mathcal{S}_n$, as these steps can be processed at the same time as the vote-step of $\mathcal{S}_n$.

4.2 Centralized orchestration

As we have seen, linear orchestration is simple and, due to its ability to abort-fast, can minimize the number of shard-steps performed to process τ. This approach comes at the cost of *consecutively* visiting each shard that has applicable vote-steps. Hence, linear orchestration takes at worst $|\text{shards}(\tau)| + 1$ consecutive consensus steps for the execution of a transaction τ. As an alternative, we can consider *parallelized orchestration* by processing all vote-steps at the same time (in parallel). Next, we propose such orchestration based on the traditional *centralized two-phase commit protocol* (Centralized-2PC) [42]. First, we present the core idea of such centralized orchestration. Then, we detail on how to efficiently collect and process the votes resulting from all vote-steps in a Byzantine environment.

Let $\mathcal{S}_r, \mathcal{S}_1, \dots, \mathcal{S}_n$ be an ordering of all shards $\mathcal{S}_r, \mathcal{S}_1, \dots, \mathcal{S}_n \in \text{shards}(\tau)$ with vote-steps. We refer to $\mathcal{S}_r$ as the *root* for τ, which will coordinate the orchestration of τ. To assure that the role of the *root* is distributed over all shards, centralized orchestration does not depend on any particular choice of $\mathcal{S}_r$. Hence, any $\mathcal{S}_r \in \text{shards}(\tau)$ will do. The root of τ starts by executing $\text{Vote}(\mathcal{S}_r)$. If $\text{Vote}(\mathcal{S}_r)$ results in a *commit* vote, then $\mathcal{S}_r$ forwards execution of τ to *all* shards $\mathcal{S}_1, \dots, \mathcal{S}_n$, after which each shard $\mathcal{S}_i$, $1 \leq i \leq n$, executes $\text{Vote}(\mathcal{S})$ *in parallel*. After forwarding, $\mathcal{S}_r$ can proceed with shard-steps of other transactions. Let $\mathcal{S}_i$, $1 \leq i \leq n$, be a shard. If $\text{Vote}(\mathcal{S}_i)$ results in a *commit* vote, then $\mathcal{S}_i$ sends a *commit vote* via cluster-sending to $\mathcal{S}_r$. Otherwise, if $\text{Vote}(\mathcal{S}_i)$ results in an *abort* vote, then $\mathcal{S}_i$ sends an *abort vote* via cluster-sending to $\mathcal{S}_r$. After sending a vote to $\mathcal{S}_r$, $\mathcal{S}_i$ can proceed with shard-steps of other transactions.

If $\mathcal{S}_r$ receives *commit* votes from each shard $\mathcal{S}_1, \dots, \mathcal{S}_n$, then τ will be committed. To do so, $\mathcal{S}_r$ forwards a *global commit vote* via cluster-sending to all shards $\mathcal{S} \in \text{shards}(\tau)$ with a commit-step $\text{Commit}(\mathcal{S})$, after which each such shard executes $\text{Commit}(\mathcal{S})$ *in parallel*. If $\mathcal{S}_r$ receives a single *abort* vote, then τ will be aborted. To do so, $\mathcal{S}_r$ forwards a *global abort vote* via cluster-sending to all shards $\mathcal{S} \in \{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ with an abort-

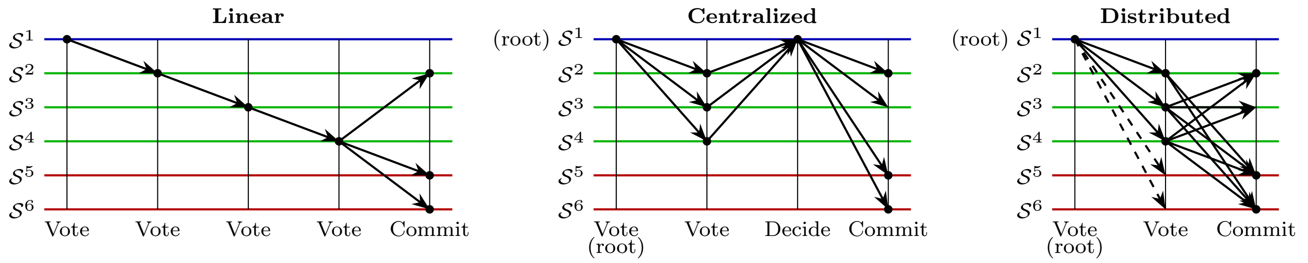

Figure 4 Two-phase commit-based orchestration of a transaction τ with shards$(\tau) = \{\mathcal{S}^1, \dots, \mathcal{S}^6\}$, in which $\mathcal{S}^1$, $\mathcal{S}^2$, $\mathcal{S}^3$, and $\mathcal{S}^4$ have vote-steps, $\mathcal{S}^2$, $\mathcal{S}^5$, and $\mathcal{S}^6$ have commit steps, and $\mathcal{S}^3$ has an abort-step. Every dot represents a single consensus step, every arrow a single cluster-sending step, and every dashed arrows a cluster-sending step used to set up distributed waiting.

step ABORT($\mathcal{S}$). All shards $\mathcal{S}$ that receive a global abort vote and voted *abort*, can ignore this vote. All shards $\mathcal{S}$ that receive a global abort vote and voted *commit*, execute ABORT($\mathcal{S}$) *in parallel*. Finally, we can eliminate the commit-step or abort-step for $\mathcal{S}_r$, as these steps can be processed at the same time as the global vote. We illustrated centralized orchestration in Figure 4, *middle*.

We notice that, in the worst case, the root $\mathcal{S}_r$ will receive $n = |\text{shards}(\tau)| - 1$ votes. For efficiency, we cannot use separate *consecutive consensus steps* at $\mathcal{S}_r$ to process each of these incoming votes: if we would use consecutive consensus steps, then receiving these n votes will take worst-case almost-as-long as the steps taken by linear orchestration to perform vote-steps at n shards in sequence. Next, we shall show that we can process these at-most $|\text{shards}(\tau)| - 1$ votes using only *a single consensus decision* at $\mathcal{S}_r$:

Lemma 1 *Let τ be a transaction and let shard $\mathcal{S}_r$ be the root that receives commit and abort votes of n other shards. Shard $\mathcal{S}_r$ will receive votes via n cluster-sending steps and can reach a commit or abort decision in at-most a single consensus step at $\mathcal{S}_r$.*

Proof Consider $\mathcal{S}_r$ receiving votes $v_1, \dots, v_n$ and let $\text{R}_1, \text{R}_2 \in \mathcal{S}_r$. Both replicas receive votes via cluster-sending and register them in some, possibly distinct, order. Independent of the order in which R_1 and R_2 receive votes, they will both receive the set of votes $\{v_1, \dots, v_n\}$, receive n_a abort votes, and n_c commit votes, $n_a + n_c = n$. Hence, eventually, R_1 and R_2 can derive the same global commit or abort decision for τ: we *do not* need to enforce a particular ordering in which votes are processed by replicas in $\mathcal{S}_r$ to agree on this decision. We still need to enforce that all replicas in $\mathcal{S}_r$ process this global abort or commit decision for τ in the same order, however. To do so, each replica in $\mathcal{S}_r$ waits until it receives all votes, after which it will use the mechanisms provided by the consensus protocol to trigger a *single consensus step* (e.g., in PBFT by forcing the primary to initiate such step) that reaches agreement on a round in which $\mathcal{S}_r$ continues processing τ (resulting in the global abort or commit decision being shared with other shards).

In a similar way, shards can process global abort votes with at-most one consensus step. Let $\mathcal{S}_i$, $1 \leq i \leq n$, be a shard. If $\mathcal{S}_i$ voted *abort*, then every replica in $\mathcal{S}_i$ is aware of this vote and can ignore the incoming global abort vote. If $\mathcal{S}_i$ voted *commit*, then every replica in $\mathcal{S}_i$ can use the mechanisms provided by the consensus protocol to reach agreement on a round in which $\mathcal{S}_i$ can execute ABORT($\mathcal{S}_i$). Finally, if a shard $\mathcal{S} \in \text{shards}(\tau)$ receives a global commit vote, then every replica in $\mathcal{S}$ can use the mechanisms provided by the consensus protocol to reach agreement on a round in which $\mathcal{S}_i$ can execute COMMIT($\mathcal{S}_i$). We conclude:

Theorem 2 *Let τ be a transaction with n_v vote-steps, n_c commit-steps, and n_a abort-steps. Using* centralized *orchestration, τ can be committed (aborted) in exactly four consecutive consensus steps using $n_v + n_c + 1$ (using $n_v + n_a + 1$) consensus steps and using $2(n_v - 1) + n_c$ (using $2(n_v - 1) + n_a$)) cluster-sending steps.*

Proof Assume that τ is committed. In this case, the root $\mathcal{S}_r$ first performs its vote-step. Then, all $n_v - 1$ other vote-steps are performed in parallel, resulting in $n_v - 1$ commit votes sent to $\mathcal{S}_r$. Next, using Lemma 1, these commit-votes are processed by $\mathcal{S}_r$ using one consensus step. Finally, as the fourth consecutive step, all n_c commit-steps are performed in parallel. Hence, we use $n_v + n_c + 1$ consensus steps, we use $(n_v - 1) + n_c$ cluster-sending steps to forward execution, and $n_v - 1$ cluster-sending steps to send commit votes. The case in which τ is aborted is analogous.

4.3 Distributed orchestration

Centralized orchestration requires *four* consecutive consensus steps. Next, we propose a method for parallelized orchestration based on the traditional *distributed two-phase commit protocol* (Distributed-2PC) [42] that only requires *three* consecutive consensus steps. We do so by instructing every shard to not just send its vote for *commit* or *abort* to the root, but instead broadcast this vote to *all* shards with either commit-steps or abort-steps.

Let $\mathcal{S}_r, \mathcal{S}_1, \dots, \mathcal{S}_n$ be an ordering of all shards $\mathcal{S}_r, \mathcal{S}_1, \dots, \mathcal{S}_n \in \text{shards}(\tau)$ with vote-steps, let $\mathcal{S}_r$ be the root for τ, let $W \subseteq \text{shards}(\tau)$ be all shards with either a commit-step or an abort-step, and let $\mathcal{S}_i$, $1 \leq i \leq n$, be a shard with a vote-step. Instead of sending the *commit* or *abort* vote resulting from $\textsc{Vote}(\mathcal{S}_i)$ to $\mathcal{S}_r$, $\mathcal{S}_i$ sends the resulting vote to *all* other shards in W. If $\mathcal{S} \in (W \cap \{\mathcal{S}_1, \dots, \mathcal{S}_n\})$ voted *abort*, then it can ignore all votes. Let $\mathcal{S}' \in W$ be a shard that did not vote abort. If $\mathcal{S}'$ has a commit-step, then it proceeds with executing $\textsc{Commit}(\mathcal{S}')$ after it receives n *commit* votes. If $\mathcal{S}'$ has an abort-step, then it proceeds with executing $\textsc{Abort}(\mathcal{S}')$ after it receives a single *abort* vote. In all other cases, $\mathcal{S}'$ can ignore the votes. We illustrated distributed orchestration in Figure 4, *right*.

To assure that each shard in W knows what to do with the votes it receives for τ, the root of τ will not only forward execution to $\mathcal{S}_1, \dots, \mathcal{S}_n$ with the instruction to vote, but also to all shards in W with the instruction to wait for votes of shards $\mathcal{S}_1, \dots, \mathcal{S}_n$ (the wait instructions also implicitly represent the commit vote of the root itself). As with the processing of votes, no consensus step is necessary at the shards in W to process these wait instructions. We conclude the following:

Theorem 3 *Let τ be a transaction with n_v vote-steps, n_c commit-steps, and n_a abort-steps. Using* distributed orchestration*, τ can be committed (aborted) in exactly three consecutive consensus steps using $n_v + n_c$ (using $n_v + n_a$) consensus steps and using $n_v(n_a + n_c) + (n_v - 1)$ (using $n_v(n_a + n_c) + (n_v - 1)$) cluster-sending steps.*

Proof Assume that τ is committed. In this case, the root $\mathcal{S}_r$ first performs its vote-step and sends its commit vote to $n_a + n_c$ shards. Next, all $n_v - 1$ other vote-steps are performed in parallel, resulting in $n_v - 1$ commit votes sent to $n_a + n_c$ shards (a total of $(n_v - 1)(n_a + n_c)$ commit votes). Finally, as the third consecutive step, each shard with a commit-step can use the techniques of the proof of Lemma 1 to process the incoming n_v commit votes and the resulting commit-step using one consensus step. Likewise, each shard with only an abort-step can ignore the commit votes without any consensus steps. Hence, we use $n_v + n_c$ consensus steps, we use $n_v - 1$ cluster-sending steps to forward execution, and $n_v(n_a + n_c)$ cluster-sending steps to send commit votes. The case in which τ is aborted is analogous.

Remark 1 We can eliminate the role of the root and reduce distributed orchestration to *two* consecutive consensus steps, this similar to how ChainSpace [1] and PCerberus [25] work. This approach requires reliable clients or recovery mechanisms to deal with faulty client behavior, however. As these recovery mechanisms have similar complexity to the *three-step* distributed orchestration we present here, we do not separately investigate such a two-step design.

5 Providing execution

Let τ be a multi-shard transaction. The second part of processing τ is to execute τ by updating any data affected by τ at the shards in $\text{shards}(\tau)$. As part of execution, one can incorporate steps to assure an *isolated* execution of τ, which makes it easier to maintain *data consistency*. Notice that single-shard steps are ordered via consensus and executed sequentially at the level of a shard. Hence, individual reads and writes always happen in full isolation, guaranteeing *write uncommitted execution* (degree 0 isolation) [4,5]. As multi-shard transactions can have several shard-steps, the processing of several multi-shard transactions can result in interleaved execution of these transactions. Hence, if further isolation is necessary for the application, then the execution method needs to incorporate some form of concurrency control. To provide concurrency control, we will describe how two-phase locking can be expressed in OEM, this without introducing additional consensus or cluster-sending steps. Using two-phase locking, ByShard provides execution with various degrees of *isolation*, e.g., *serializable execution* (degree 3), *read committed execution* (degree 2), and *read uncommitted execution* (degree 1) [4,5,22]. As a baseline, we also describe two basic lock-free execution methods that only provide degree 0 isolation.

To illustrate execution, we formalize the *account-transfer* data and transaction model of preceding examples. For this purpose, we assume that each transaction τ is a pair (C, M) in which C is a set of constraints of the form

$$\textsc{Con}(\mathrm{X}, \mathrm{y}) = \text{“the balance of } X \text{ is at least } y\text{”}$$

and M a set of modifications of the form

$$\textsc{Mod}(\mathrm{X}, \mathrm{y}) = \text{“add } y \text{ to the balance of } X\text{”}.$$

We write $C(\mathcal{S})$ and $M(\mathcal{S})$ to denote the constraints and modifications in C and M, respectively, that affect accounts maintained by $\mathcal{S}$. Semantically, a system *commits* to τ only if all constraints in C hold, in which case all modifications in M are applied to the system. Notice that these minimalistic account-transfer transactions are sufficient to represent all transactions in preceding examples. In Section 7, we discuss why this minimalistic account-transfer data and transaction model is representative for general-purpose workloads for resilient data management systems.

5.1 Isolation-free direct execution

First, we propose a basic execution method with minimal isolation by formalizing the *isolation-free execution method* employed in the linearly orchestrated processing plan of Example 5.

Let $\tau = (C, M)$ be a transaction with $\mathcal{S} \in \text{shards}(\tau)$. Shard $\mathcal{S}$ needs a vote-step whenever constraints need to be checked at $\mathcal{S}$ ($C(\mathcal{S}) \neq \emptyset$). This vote-step σ checks whether all constraints in $C(\mathcal{S})$ hold. If these constraints hold, then σ makes a commit vote. Otherwise, σ makes an abort vote. To avoid a separate commit-step for τ at $\mathcal{S}$, we optimistically assume that τ will not abort and let the vote-step σ perform all modifications in $M(\mathcal{S})$ after it voted commit. When the transaction gets aborted, we need to roll back any modifications made by σ. Hence, if $M(\mathcal{S}) \neq \emptyset$, we also construct an abort-step $\textsc{Abort}(\mathcal{S})$ that rolls back all modifications in $M(\mathcal{S})$ by performing the modifications $\{\textsc{Mod}(\text{X}, \text{-y}) \mid \textsc{Mod}(\text{X}, \text{y}) \in M(\mathcal{S})\}$.

If $\mathcal{S}$ only has modifications ($C(\mathcal{S}) = \emptyset$), then $\mathcal{S}$ only needs a commit-step that performs all modifications in $M(\mathcal{S})$.

The main strength of isolation-free execution is the minimal amount of shard-steps it produces: if a transaction is committed, then each shard will only execute a single shard-step (a vote-step if there are constraints, a commit-step otherwise). Unfortunately, isolation-free execution provides only degree 0 isolation, which can lead to violations of constraints on the data in many applications:

Example 7 Consider the sharded banking example of Example 4. Assume that the system does not allow negative account balances and consider transactions

$$\begin{aligned}\tau_1 &= \textsc{Con}(\text{A}, 100), \textsc{Con}(\text{B}, 700),\\ &\qquad \textsc{Mod}(\text{A}, 400), \textsc{Mod}(\text{B}, \text{-}400);\\ \tau_2 &= \textsc{Con}(\text{A}, 500), \textsc{Mod}(\text{A}, \text{-}300), \textsc{Mod}(\text{E}, 300),\end{aligned}$$

and their isolation-free linearly orchestrated execution illustrated in Figure 5. As one can see, the balance of A will become negative, breaking the constraint put in place. This is caused by operation $\textsc{Con}(\text{A}, 500)$ of τ_2, which performs a so-called *dirty read* [5, 42].

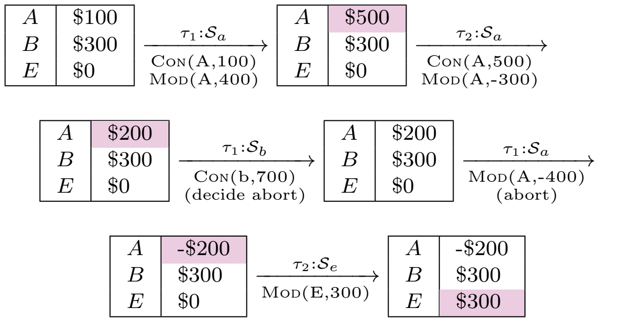

Figure 5 Evolution of the *current state* while step-wise executing the transactions of Example 7.

As isolation-free execution provides minimal isolation, it is unable to prevent phenomena such as dirty reads that can lead to data inconsistencies. Isolation-free execution does provide *atomicity*, however: either *all* or *none* of the modifications of a transaction are permanent. One way to deal with constraint violations such as in Example 7 is by assuring that roll backs do not invalidate constraints in a *domain-specific* manner. To illustrate this, assume we want to assure that accounts never have negative balances.

On the one hand, rolling back $\textsc{Mod}(\text{X}, \text{y})$ with $y \leq 0$ (a removal) will *increase* the balance of X and, hence, will never make the balance of X negative. Consequently, these modifications are *safe*. Furthermore, notice that if $\textsc{Con}(\text{X}, \text{-y})$ and $\textsc{Mod}(\text{X}, \text{y})$, $y \leq 0$, are part of a single vote-step, then they are executed in isolation as a single unit and, hence, the modification will never make the balance negative (this pattern of constraint checking and removing of balance can be seen as a *lock on available resources*, whereas rolling back the removal is a *release of unused resources*).

On the other hand, rolling back $\textsc{Mod}(\text{X}, \text{y})$ with $y \geq 0$ (an addition) will *decrease* the balance of X and, hence, can make the balance of X negative. Consequently, these modifications are *unsafe*. To assure that unsafe modification do not invalidate constraints, one can perform these modifications when *committing* (via a commit-step). This means that, in the worst case, every affected shard must execute *two* shard-steps when committing: the vote-step checks constraints and performs safe modifications (which, on abort, are rolled back via the abort-step) and the commit-step performs unsafe modifications. We refer to this execution method, in which safe modifications are part of vote-steps and unsafe modifications are part of commit-steps (hence, executed safely), as *safe isolation-free execution*.

5.2 Lock-based execution

Although safe isolation-free execution is able to maintain some data consistency, it does so in an domain-specific manner that cannot be applied to all situations. As a more general-purpose method towards maintaining data consistency, we can enforce higher isolation levels for transaction processing, e.g., *degree 3* (serializable execution). The standard way to do so in a multi-shard environment is by using *two-phase locking* [4, 42]. First, we describe the working of two-phase locking. Then, we discuss how to implement two-phase locking with minimal coordination in a Byzantine environment.

Consider a multi-shard transaction τ. When executing τ, τ needs to obtain a *read lock* on each data item D before it reads D and a *write lock* on each data item D before it writes D. Several transactions can hold a read lock on D at the same time, while write locks on D are exclusive: if τ', $\tau' \neq \tau$, holds a write lock on D, then τ cannot obtain any locks on D, and if τ', $\tau' \neq \tau$, holds a read lock on D, then τ cannot obtain a write lock on D, but can obtain a read lock on D. When τ cannot obtain a lock on D that it needs, it simply waits until previous transactions finish and release their locks on D. To provide *serializability*, τ is barred from obtaining new locks after releasing any locks: this assures that there is a point in time where τ is the only transaction that holds all write locks on data items affected by τ, at which point τ can make any changes to these data items in an indivisible atomic manner.

To avoid deadlocks when using these *blocking locks*, we enforce that each transaction locks data items in exactly the same order [42]. To minimize the number of shard-steps, we assume a fixed order on shards and on data items within shards, and obtain all locks in that order. Consequently, ByShard requires linear orchestration when using blocking locks.

Example 8 Consider the sharded banking example of Example 4 and the transaction

$\tau =$ "if *Ana* has \$500 and *Bo* has \$300 then
move \$200 from *Ana* to *Ben*".

Assume that shards are ordered as $\mathcal{S}_a, \dots, \mathcal{S}_z$ and that accounts are ordered on account holder name. To execute this transaction, we first obtain a write lock on the account of Ana in $\mathcal{S}_a$, then a write lock on the account of Ben in $\mathcal{S}_b$, and, finally, a read lock on the account of Bo in $\mathcal{S}_b$.

Let $\tau = (C, M)$ be a transaction, let $\mathcal{S} \in \text{shards}(\tau)$, and let $\textsc{Accounts}(\mathcal{S})$, defined by

$$\{X \mid \textsc{Con}(\mathrm{X}, \mathrm{y}) \in C(\mathcal{S}) \lor \textsc{Mod}(\mathrm{X}, \mathrm{y}) \in M(\mathcal{S})\},$$

be the set of accounts affected at $\mathcal{S}$. During the vote-step $\textsc{Vote}(\mathcal{S})$, we acquire a lock $\textsc{Lock}(X)$ for every account $X \in \textsc{Accounts}(\mathcal{S})$ in some predetermined order. If there is a $\textsc{Mod}(\mathrm{X}, \mathrm{y}) \in M(\mathcal{S})$, then we acquire a write lock for X. Otherwise, we acquire a read lock. After acquiring the lock on X, we check any constraint $\textsc{Con}(\mathrm{X}, \mathrm{y}) \in C(\mathcal{S})$. If a constraint does not hold, then $\textsc{Vote}(\mathcal{S})$ votes abort and releases all locks already acquired in $\mathcal{S}$. We purposely check these constraints as soon as possible to minimize the amount of time locks are held. Otherwise, if all constraints hold, then $\textsc{Vote}(\mathcal{S})$ votes commit. Next, the commit-step $\textsc{Commit}(\mathcal{S})$ performs all modifications $M(\mathcal{S})$ followed by performing $\textsc{Release}(X)$ to release all locks in $\mathcal{S}$ for all accounts $X \in \textsc{Accounts}(\mathcal{S})$. Finally, the abort-step $\textsc{Abort}(\mathcal{S})$ performs $\textsc{Release}(X)$, for all accounts $X \in \textsc{Accounts}(\mathcal{S})$, to release all locks in $\mathcal{S}$. We have:

Theorem 4 *Let τ be a transaction with $n = |\text{shards}(\tau)|$. To process τ using two-phase locking, we need n vote-steps to obtain all locks, followed by $n-1$ commit-steps or abort-steps to release all locks. Hence, τ can be processed using $2n-1$ consensus steps and $2n-2$ cluster-sending steps.*

Proof To prove the theorem, we only need to prove that vote-step $\textsc{Vote}(\mathcal{S})$ of shard $\mathcal{S} \in \text{shards}(\tau)$ can obtain all its locks using only a single consensus step at $\mathcal{S}$. Execution of $\textsc{Vote}(\mathcal{S})$ starts after $\mathcal{S}$ reached consensus on this step, and we will prove that no further consensus steps for $\textsc{Vote}(\mathcal{S})$ are required. Let $\textsc{Vote}(\mathcal{S}) = \{\dots, \textsc{Lock}(X), \dots\}$. During execution of $\textsc{Vote}(\mathcal{S})$, we distinguish two possible cases:

1. The lock on X *can* be obtained, in which case execution of $\textsc{Vote}(\mathcal{S})$ continuous.
2. The lock on X *cannot* be obtained. In this case, execution of $\textsc{Vote}(\mathcal{S})$ needs to wait until the lock on X can be obtained. To do so, as part of the execution of $\textsc{Vote}(\mathcal{S})$, every replica $\mathrm{R} \in \mathcal{S}$ puts $(\tau, \textsc{Vote}(\mathcal{S}))$ on a wait-queue $Q_{\mathrm{R}}(X)$.

Let $\mathrm{R}_1, \mathrm{R}_2 \in \mathcal{S}$. We assume that wait-queues $Q_{\mathrm{R}_1}(X)$ and $Q_{\mathrm{R}_2}(X)$ operate *deterministic*: if the same operations are applied to $Q_{\mathrm{R}_1}(X)$ and $Q_{\mathrm{R}_2}(X)$, then the queues always yield the same results. Now consider the case in which the lock on X cannot be obtained. Let τ', $\tau \neq \tau'$, be the transaction that is holding the lock on X and let $\sigma = \{\dots, \textsc{Release}(X), \dots\}$ be the commit-step or abort-step of τ' for shard $\mathcal{S}$. During execution, shard-step σ will release the lock on X. When doing so, each replica $\mathrm{R} \in \mathcal{S}$ wakes up transactions in $Q_{\mathrm{R}}(X)$ for execution directly after shard-step σ. We distinguish two cases:

1. The next transaction in $Q_{\mathrm{R}}(X)$ wants to obtain a read lock, while τ' held a write lock. In this case, wake up all transactions in queue order in $Q_{\mathrm{R}}(X)$ that want to obtain a read lock (all these transactions can hold the *non-exclusive* read lock at the same time).
2. The next transaction in $Q_{\mathrm{R}}(X)$ wants to obtain a write lock. If τ' was the last transaction holding any lock on X, then we wake up the next transaction (as this transaction requires an *exclusive* write lock).

This wake up step is part of the deterministic execution of σ and wake-up queues operate deterministic. Hence, *no* consensus steps are necessary to determine which transactions need to be executed next and to initiate execution of these next transactions.

We notice that we cannot always minimize the number of affected shards while processing τ via two-phase locking:

Example 9 Consider the sharded banking example of Example 8 and the transaction τ = "if *Bo* has \$500, then move \$200 from *Bo* to *Ana*". Due to the ordering on shards and accounts used, we *always first* need to obtain a write lock on the account of Ana (in shard $\mathcal{S}_a$) before we can inspect the balance of Bo (in shard $\mathcal{S}_b$), even if Bo does not have sufficient balance. This is in contrast with the isolation-free execution methods, as these methods can *first* inspect the balance of Bo and directly abort execution (without touching shard $\mathcal{S}_a$).

Locking and other isolation levels The strength of two-phase locking is that it provides serializability. The downside is that it can cause large transaction processing latencies whenever *contention* is high:

Example 10 Consider a system in which consensus steps take $t = 30$ ms each, while all other steps take negligible time (see Example 1). We consider transactions τ_1 and τ_2 such that τ_1 writes to data items $D_1, \ldots, D_{10}$ that are held in shards $\mathcal{S}_1, \ldots, \mathcal{S}_{10}$, respectively, while τ_2 only writes to data item D_1. Transaction τ_1 executes first at $\mathcal{S}_1$ and obtains the write lock on D_1. Next, τ_2 executes at $\mathcal{S}_1$, cannot obtain the write lock on D_1, and has to wait until τ_1 finishes execution and releases the lock on D_1. To do so, τ_1 has to first obtain locks in $m-1$ shards, after which it can return to $\mathcal{S}_1$ to release the lock on D_1. Hence, τ_1 has to perform m consecutive consensus steps. Even if τ_1 can obtain the locks on $D_2, \ldots, D_{10}$ immediately, it will take at least $10t = 300$ ms before τ_2 can resume execution, even though the actual execution of τ_2 would only take $t = 30$ ms.

One way to partially deal with Example 10 is by not imposing degree 3 isolation (serializable execution), and the primitives we propose to provide degree 3 isolation can easily be used to provide lower levels of isolation [4, 5, 22]. For example:

1. in *read uncommitted execution* (degree 1 isolation), no read locks are obtained on any data item (while write locks are used in the usual way), thereby reducing lock contention sharply for read-heavy workloads; and
2. in *read committed execution* (degree 2 isolation), read locks on each data item D are released directly after reading D (while write locks are used in the usual way), thereby minimizing the time read locks are held.

Non-blocking locks Using lower isolation levels only partially mitigates the issues illustrated in Example 10. To further deal with this, one can opt to replace *waiting* by *failing*: whenever a lock cannot be obtained by a transaction τ, τ aborts. This approach guarantees that processing latencies of transactions and resource utilization at the replicas are kept in check in periods of high contention, this at the cost of aborted transactions that could otherwise be successfully executed. As these *non-blocking locks* will never cause deadlocks, these locks can be obtained in any order, enabling their usage in combination with *all* orchestration methods.

6 Performance evaluation

In the previous sections, we introduced ByShard as a framework for sharded resilient systems. As part of this framework, we also presented general-purpose methods by which ByShard can orchestrate and execute multi-shard transactions. Combining these methods results in *eighteen* multi-shard transaction processing protocols that each make their own trade-offs between performance, isolation level, latency, and abort rates. Furthermore, protocols used by contemporary sharded resilient systems such as AHL [11] and Chainspace [1] can also easily be expressed within the orchestrate-execute model of ByShard. We refer to Table 1 for an analytical comparison of each of these *twenty* protocols.

Remark 2 In practical deployments of ByShard, end-users only need to use one of these eighteen multi-shard transaction processing protocols. In our experiments, we use such single-protocol deployments, as we are interested in the differences between the protocols. This does not rule out deployments of ByShard that use several protocols simultaneously: ByShard does support the usage of several protocols at the same time such that users can select the appropriate isolation level for individual transactions.

Table 1 Overview and comparison of the *eighteen* multi-shard transaction processing protocols of ByShard and of the multi-shard transaction processing protocols of AHL [11] and Chainspace [1].

					Steps of the Transaction[a]				Complexity to Commit[b]		
Orchestration	Protocol	Isolation[c]	Safety[d]	Abort-Fast[5]	Vote	Commit	Abort	Locks	Shard-Steps	(*sequential*)	Votes[e]
***I**solation-**F**ree execution* (write uncommitted), **u**nsafe.											
Linear	LIFu	Degree 0	Unsafe	Abort-fast	$c+b$	m	b	(none)	n	$c+b+1$	(none)
Centralized	CIFu	Degree 0	Unsafe	At the root	$c+b+1$	m	b	(none)	$n+1$	4	$(c+b)+(b+m)$
Distributed	DIFu	Degree 0	Unsafe	At the root	$c+b$	m	b	(none)	n	3	$(c+b)\cdot(b+m)$
***I**solation-**F**ree execution* (write uncommitted), **s**afe.											
Linear	LIFs	Degree 0	Safe	Abort-fast	$c+b$	$b+m$	b	(none)	$n+b-1$	$c+b+1$	(none)
Centralized	CIFs	Degree 0	Safe	At the root	$c+b+1$	$b+m$	b	(none)	$n+b+1$	4	$(c+b)+(b+m)$
Distributed	DIFs	Degree 0	Safe	At the root	$c+b$	$b+m$	b	(none)	$n+b$	3	$(c+b)\cdot(b+m)$
*Lock-based execution, **R**ead-**U**ncommitted, **b**locking.*											
Linear	LRUb	Degree 1	Safe	Abort-fast	n	$b+m$	$b+m$	$b+m$	$n+b+m-1$	$n+1$	(none)
*Lock-based execution, **R**ead-**U**ncommitted, **n**on-**b**locking.*											
Linear	LRUnb	Degree 1	Safe	Abort-fast	n	$b+m$	$b+m$	$b+m$	$n+b+m-1$	$n+1$	(none)
Centralized	CRUnb	Degree 1	Safe	At the root	$n+1$	$b+m$	$b+m$	$b+m$	$n+b+m$	4	$n+(b+m)$
Distributed	DRUnb	Degree 1	Safe	At the root	n	$b+m$	$b+m$	$b+m$	$n+b+m$	3	$n\cdot(b+m)$
*Lock-based execution, **R**ead-**C**omitted, **b**locking.*											
Linear	LRCb	Degree 2	Safe	Abort-fast	n	$b+m$	$b+m$	n	$n+b+m-1$	$n+1$	(none)
*Lock-based execution, **R**ead-**C**omitted, **n**on-**b**locking.*											
Linear	LRCnb	Degree 2	Safe	Abort-fast	n	$b+m$	$b+m$	n	$n+b+m-1$	$n+1$	(none)
Centralized	CRCnb	Degree 2	Safe	At the root	$n+1$	$b+m$	$b+m$	n	$n+b+m$	4	$n+(b+m)$
Distributed	DRCnb	Degree 2	Safe	At the root	n	$b+m$	$b+m$	n	$n+b+m$	3	$n\cdot(b+m)$
*Lock-based execution, **S**erializable, **b**locking.*											
Linear	LSb	Degree 3	Safe	Abort-fast	n	n	n	n	$2n-1$	$n+1$	(none)
*Lock-based execution, **S**erializable, **n**on-**b**locking.*											
Linear	LSnb	Degree 3	Safe	Abort-fast	n	n	n	n	$2n-1$	$n+1$	(none)
Centralized	CSnb	Degree 3	Safe	At the root	$n+1$	n	n	n	$2n$	4	$n+(n-1)$
Distributed	DSnb	Degree 3	Safe	At the root	n	n	n	n	$2n$	3	$n\cdot(n-1)$
Lock-based execution, Serializable, non-blocking.											
Ref. committee	AHL [11]	Degree 3	Safe	(none)	$n+2$	n	n	n	$2n+2$	4	$n+n$
Distributed[f]	Chainspace [1]	Degree 3	Safe	(none)	n	n	n	n	$2n$	2	$n\cdot n$

[a] The complexity of the transaction in terms of the number of shard-steps (vote-steps, commit-steps, and abort-steps) and the number of shard-steps that will attempt to obtain locks. In specific, for a transaction that affects $n = c+m+b$ shards, of which c have only checks, m have only modifications, and b have both checks and modifications, we specify the maximum number of vote-steps, commit-steps, and abort-steps. For clarity, we have omitted minor optimizations from this overview, e.g., in all protocols one can merge a limited number of commit- and abort-steps with vote-steps.

[b] The cost in terms of the number of shard-steps (consensus steps) and votes (via cluster-sending) to commit a transaction that affects $n = c+m+b$ shards.

[c] The degree of transaction isolation provided by the protocol [4].

[d] We say that a protocol is safe if it is able to maintain the constraint "no account should have a negative balance".

[e] A protocol is *abort-fast* whenever an abort decision in a single vote-step will directly trigger a global abort and prevent any further vote-steps.

[f] Chainspace uses a variant of distributed orchestration that relies on the client to initiate multi-shard transaction processing. We refer to Remark 1 for further details.

To gain further insight in the performance attainable by sharded resilient systems, we implemented the ByShard framework, the orchestrate-execution model, and the eighteen multi-shard transaction processing protocols obtained from the presented orchestration an execution methods. For comparison, we also implemented the protocol of AHL [11], which has a novel design that is most similar to the design of our *Centralized, Serializable, non-blocking* protocol CSnb, the main difference being that AHL uses a dedicated *reference committee* to coordinate processing of multi-shard transactions, whereas in CSnb each transaction is coordinated by a root-shard chosen from the set of shards affected by that transaction. Our implementation of AHL is granted a dedicated extra shard for use as the reference committee. Finally, we note that the design of our *Distributed, Serializable, non-blocking* protocol DSnb is a generalization of the design of Chainspace [1]. We refer to Remark 1 for further details on the relationship between the three-step design of DSnb and the two-step design of Chainspace.

Next, we deployed our implementation on a simulated sharded resilient system. In specific, we abstract the operations of *consensus* and *cluster-sending*, while deploying full shards that execute all replica-specific operations necessary for transaction orchestration and execution. This deployment provides detailed control over consensus and cluster-sending costs, enables fine-grained measurements of performance metrics, and allows us to deploy on hundreds of shards.[2]

6.1 Experimental Setup

We run experiments in which we measured the behavior of the system as a function of eight distinct parameters. The details on these eight experiments can be found in Section 6.2. In each experiment, we run a workload of 5000 transactions. Unless specified otherwise, each transaction affects 16 distinct accounts by putting constraints on 8 accounts (read operations), removing balance from 4 accounts (write operations), and adding balance to 4 accounts (write operations). The accounts affected by these operations are chosen uniformly at random from a set of active accounts. Each account on each shard starts with an initial balance of 2000 and transactions add or remove 500 balance per modification (on average, these are chosen via a binomial distribution with $n = 1000$ and $p = 0.5$). We run experiments with 64 shards and 8192 active accounts (128 active accounts per shard). Finally, the experiments are set up such that cluster-sending takes 10 ms and consensus decisions take at-least 30 ms. To take into account contention at individual shards, each shard can perform up-to 1000 decisions/s (we assume a consensus protocol with out-of-order processing, but consensus decisions start consecutively).

The *number of active accounts* is low to increase contention and the *number of affected accounts* per transaction is high to maximize complexity. This is on purpose: in our experiments, we want to study how the multi-shard transaction processing protocols we compare differ in their operations and we are especially interested in the performance of the system when dealing with *multi-shard transactions* that require substantial coordination to deal with contention. Indeed, in workloads with low contention (e.g., more active accounts), locking has no discernible side-effects on the performance or behavior of the system. Furthermore, in workloads that mainly consist of *single-shard transactions*, each of the multi-shard transaction protocols we look at will fall back to the same underlying single-shard consensus protocol to effectively process such single-shard transactions. We refer to Section 2 for details on how single-shard transactions are processed. In each experiment, we collected the following detailed measurements:

- The *total runtime* represents the elapsed real time to process the workload.
- The *cumulative duration* represents the sum of the transaction duration (the elapsed real time to process that transaction) of each transaction in the workload.[3]
- The *average throughput* represents the average number of transactions processed per second.
- The *average committed throughput* represents the average number of transactions committed per second.
- The *committed transactions* represent the number of transactions that are committed.[4]
- The *constraint failures* represent the total number of constraint checks that did not hold.[5]

[2] The full C++ implementation of these experiments and the raw measurements are available at https://www.jhellings.nl/projects/byshard/.

[3] The transaction duration includes waiting times (e.g., waiting for locks to be released, waiting for votes to arrive, waiting for a next shard-step to be executed). As many transactions can be active in parallel (even at a single shard due to waiting), the cumulative duration can be much higher than the product of the number of shards and the total runtime.

[4] All other processed transactions are aborted (either due to constraint failure or, when non-blocking locks are used, the inability to acquire locks).

[5] Linear orchestration can only have a single constraint failure per transaction (which will lead to an abort for that transaction), while both centralized and distributed orchestration can have many constraint failures per transaction (which will lead to a single abort for that transaction).

- The *median shard-steps* represent the median number of shard-steps (each representing a single consensus step) performed per shard.
- The *shard-step imbalance* represent the maximum difference between the number of shard-steps performed by any two shards.
- The *total locks* represents the total number of attempts to obtain a read or write lock.
- The *failed locks* represent the total number of failed attempts to obtain a read or write lock.
- The *total votes* represent the total number of commit and abort votes casts.

6.2 Experimental Details

Next, we will detail the *eight* experiments that we performed and, per experiment, provide the main findings.

The scalability experiment In our first experiment, we study the impact of *sharding* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of shards* while keeping all other parameters the same (including the workload and the initial dataset). Increasing the number of shards will increase the available parallel processing power, while decreasing the number of accounts per shard. Hence, we increase the average number of multi-shard transactions and the number of shards affected by each transaction. The results of this experiment can be found in Figure 6.

The results show that all protocols have excellent scalability: when the number of shards is increased, the median amount of work per shard (consensus steps and vote processing steps) decreases rapidly. This is especially the case when moving beyond 16 shards, as each transaction will affect 16 accounts at at-most 16 distinct shards. Furthermore, all our eighteen multi-shard transaction protocols show a good distribution of consensus steps among all shards, as the imbalance in steps is relatively small compared to the median steps per shard.

Although the imbalance in steps is small, there is a noticeable imbalance in shard-steps for the protocols that use linear orchestration. This an unfortunate side-effect of to the deterministic order in which vote-steps are performed in protocols that use linear orchestration: in these protocols, shard-steps are executed consecutively (see, e.g., Theorem 1) using some deterministic shard-ordering. The shards that appear early in this order will have more work than the other shards (this is especially when many transactions abort-fast), causing the observed moderate imbalance. Furthermore, we notice that this side-effect cannot be avoided for protocols that use blocking locks (without further measures to prevent deadlocks).

When comparing protocols that use the same execution method and only differ in orchestration method, we see that the protocol using distributed orchestration has the lowest runtime and highest throughput, followed by the protocol using centralized orchestration, followed by the protocol using linear orchestration. When comparing protocols that use the same orchestration method and only differ in execution method, we see that the protocols using execution methods that provide lower degrees of isolation have the lowest runtime and duration, and, consequently, the highest throughput.

Furthermore, the experiment underlines the benefits and drawbacks of parallel processing of shard-steps in protocols that use the centralized and distributed orchestration methods. On the one hand, these *parallel* protocols have lower runtimes and transaction durations than their linear counterparts, even though parallel protocols perform many more steps. As a consequence, parallel protocols typically are able to reach higher throughputs than their linear counterparts. On the other hand, the negative effects of contention are higher in parallel protocols than in their linear counterparts. This results in higher rates of constraint failures and, when lock-based execution is used, higher rates of failed locks. These negative effects of contention reduce the number of committed transactions (especially when non-blocking lock-based execution is used). The high throughput of parallel protocols can offset the negative effects parallel processing has on contention, however: the average committed throughput is higher for protocols that use distributed orchestration than for their linear counterparts, even though protocols that use distributed orchestration also have the highest abort rates.

In line with the original evaluation of AHL [11, Section 7.3], we see that the reference committee of AHL is a bottleneck for multi-shard transaction processing: the multi-shard transaction processing performance of AHL is determined by the time it takes for the reference committee to perform its orchestration tasks. Due to the high amount of multi-shard transactions in our workload, the usage of a reference committee causes a large imbalance in the number of shard-steps performed by the reference committee and by other shards. Due to these observations, the scalability of AHL for multi-shard transaction workloads is limited. As AHL relies on the reference committee, transactions are less likely to content for the same locks, however. Consequently, the impact of contention is lower in AHL than in similar protocols in ByShard, e.g., the protocols that use

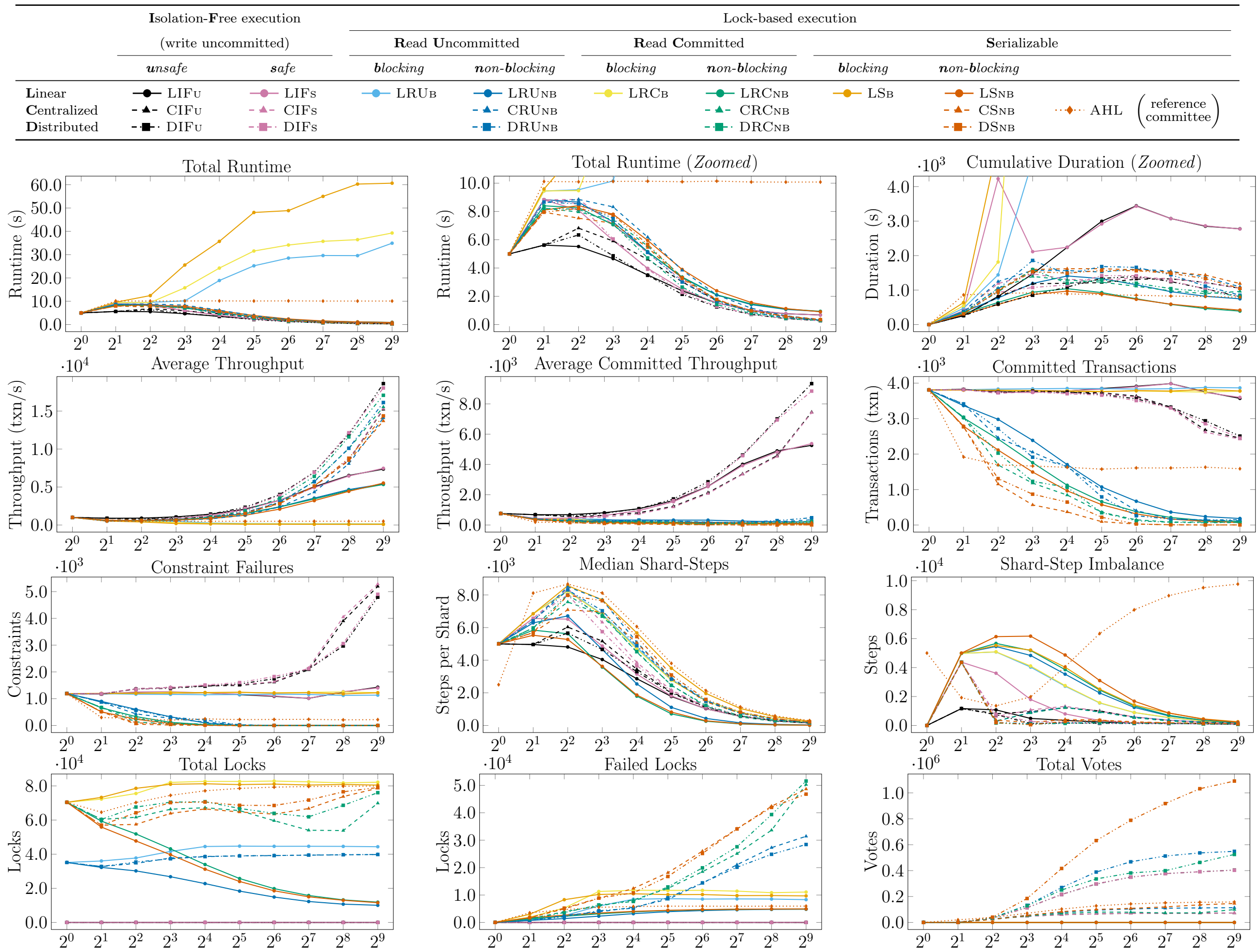

Figure 6 Measurements for the *scalability experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *number of shards*. We used a fixed dataset with 8192 active accounts and a fixed workload of 5000 transactions (each affecting 16 accounts).

centralized orchestration, due to which AHL has lower abort rates than the centralized protocols of ByShard.

The contention experiment In our second experiment, we study the impact of *contention* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of active accounts per shard*. For each case, we generate appropriate workloads as a function of the number of active accounts in the system. Increasing the number of active accounts decreases the probability that two transactions affect the same account and, hence, decreases contention. The results of this experiment can be found in Figure 7.

The results further underline the large impact contention has on parallel protocols: we see that a decrease in contention always causes a decrease in constraint failures and lock failures. This effect is sharpest in the parallel protocols. Consequently, parallel protocols see the strongest improvement in the number of committed transactions and the average committed throughput when contention decreases. Finally, we see that protocols that use lock-based execution with blocking locks, which have exceptionally high commit rates in all cases, have good runtimes and excellent scalable performance whenever contention is low.

The factor-scalability experiment In our third experiment, we study the impact of *scaling the system* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of shards* and, as we keep the number of active accounts per shard constant, the *number of active accounts*. For each case, we generate appropriate workloads as a function of the number of accounts in the system. As in the *scalability experiment*, increasing the number of shards increases the available parallel processing power. Furthermore, increasing the number of accounts decreases contention. The results of this experiment can be found in Figure 8.

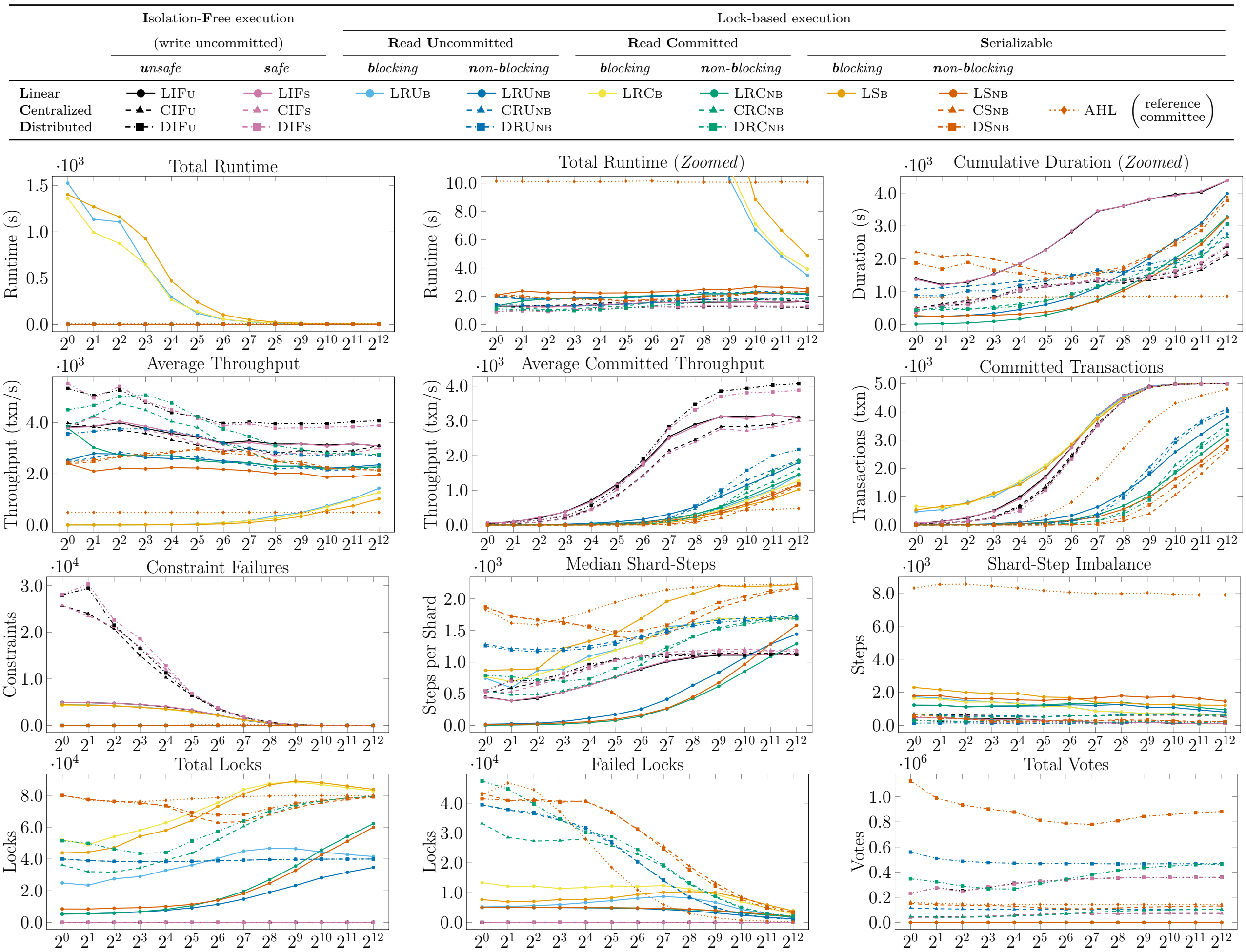

Figure 7 Measurements for the *contention experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *number of accounts per shard*. We used 64 shards and a workload of 5000 transactions (each affecting 16 accounts).

The results underline the findings of the previous two experiments. Furthermore, we see that scaling up the system by increasing the number of shards that each hold a constant amount of active accounts sharply increases transaction throughput and committed transaction throughput, as scaling the system sharply reduces the amount of work per shard and—due to the increase in accounts—the contention (and its negative effects on the amount of committed transactions).

The account skew experiment We inspect the impact of *skew* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *skew in the accounts affected* by the transactions in the workload, while keeping all other parameters the same (including the number of shards and the initial dataset). For each case, we generate appropriate workloads in which accounts affected by the transactions in the workload are chosen via a geometric distribution with $p = f/8192$, where $f \in \{2, \ldots, 10\}$ is the skew factor (instead of a uniform distribution). In practice, this implies that the most likely account has probability $f/8192$ to be chosen (instead of $1/8192$). The results of this experiment can be found in Figure 9.

As higher skew causes higher contention, the results show that increasing the skew increases the negative impacts of contention, especially when blocking locks are used. Consequently, the results of this experiment underline the observations made in the other experiments in the paper. Based on the results, we make the following final observations: we see that the negative impacts of skew are highest in protocols that use centralized orchestration or distributed orchestration, especially when looking at cummulative duration. We believe this is due to the increase in shard-step imbalance when increasing skew, which is due to more transactions involving shards that hold the accounts with the highest skew. Consequently, the protocols that use linear orchestration are able to outperform their cen-

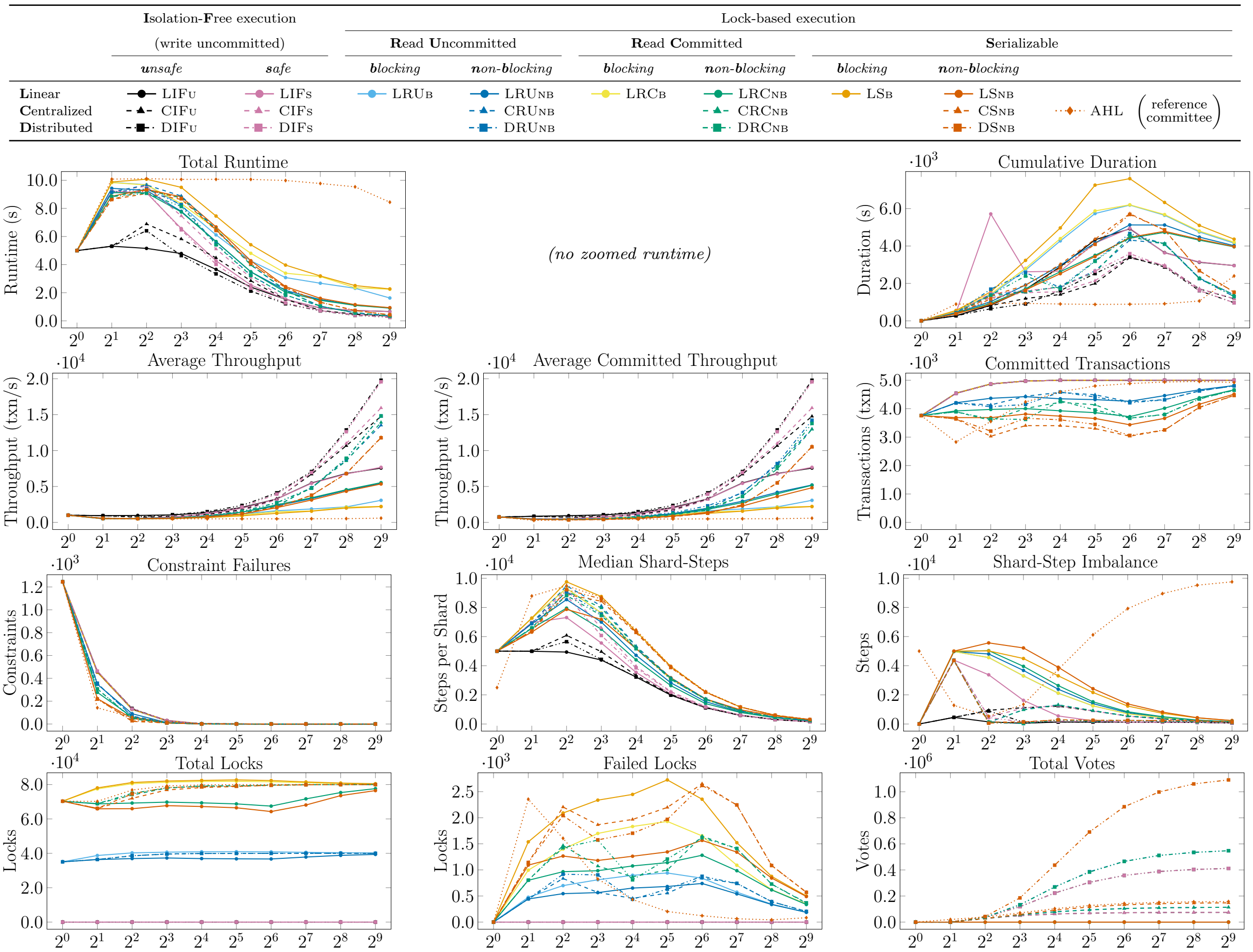

Figure 8 Measurements for the *factor-scalability experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *scalability-factor* by which the number of accounts and shards grow. We used a workload of 5000 transactions (each affecting 16 accounts).

tralized and distributed counterparts when the skew is sufficiently high.

The local-global experiment In our fifth experiment, we study the impact of *multi-shard transactions* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *percentage of global (multi-shard) transactions* in the workload while keeping all other parameters the same (including the number of shards and the initial dataset). For each case, we generate appropriate workloads that mix local (single-shard) transactions with the required number of global transactions. Local transactions are generated such that they affect a consecutive set of 16 accounts in their local shard. The results of this experiment can be found in Figure 10.

The results show that the cost of coordinating multi-shard transactions (e.g., in terms of steps, failed locks, and votes) scales proportional with the percentage of multi-shard transactions (that affect more shards and, hence, perform more steps than local transactions). Consequently, increasing the percentage of multi-shard transactions generally decreases performance. At the same time, the experiment confirms that AHL has excellent scalability on workloads with a high amount of single-shard transactions.

The constraint failure experiment In our sixth experiment, we inspect the impact of *constraint failures* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *initial balance*, while keeping all other parameters the same (including the number of shards and the workload). By increasing the initial balance of accounts, we increase the number of transactions that can successfully remove balance from accounts and, hence, decrease the number of transactions that will experience constraint failure. Consequently, an increase in initial balance increases the number of transactions that can commit. The results of this experiment can be found in Figure 11.

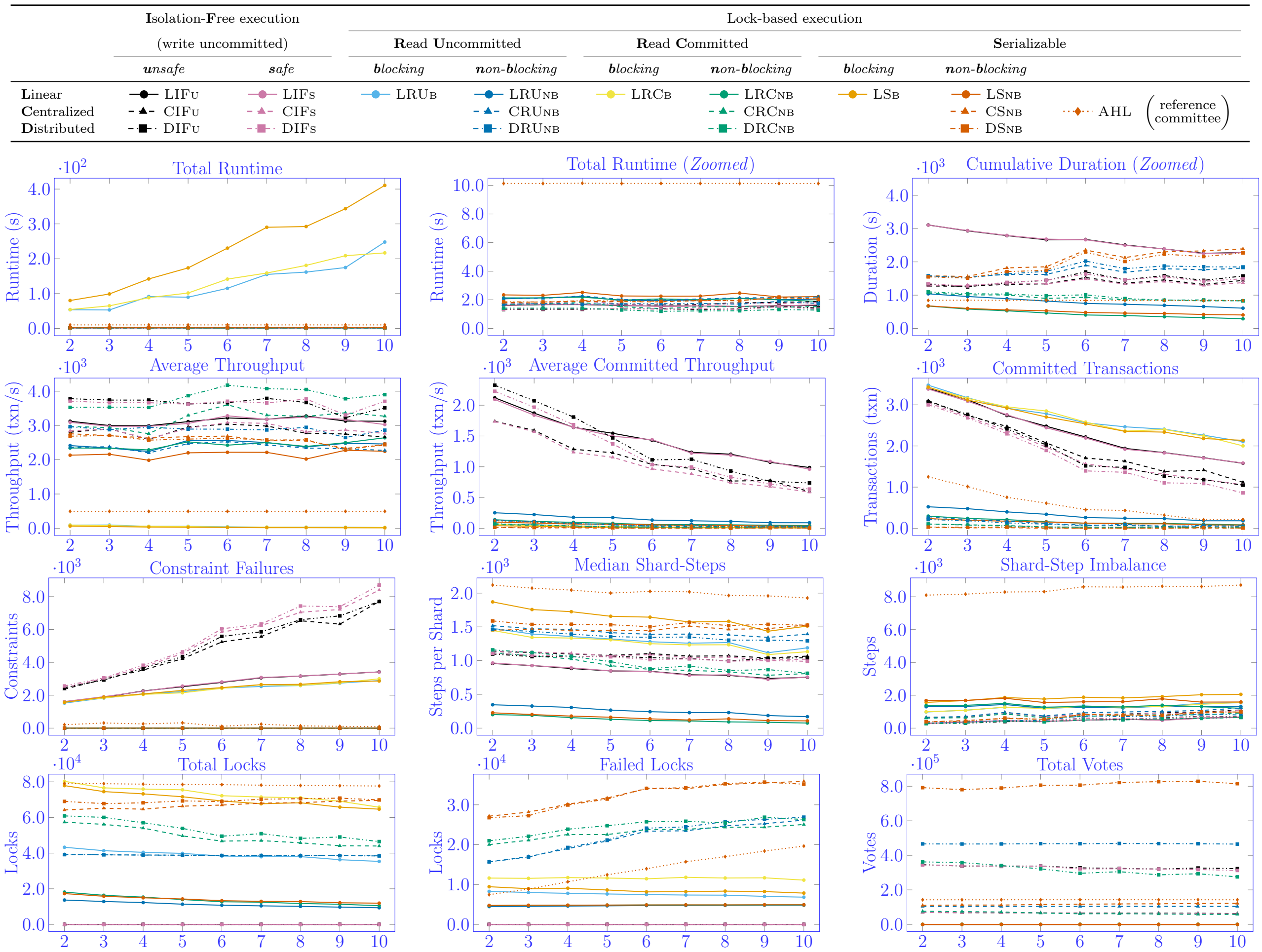

Figure 9 Measurements for the *account skew experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *skew-factor of transactions*. We used a fixed initial dataset with 8192 active accounts, 64 shards, and a workload of 5000 transactions (each affecting 16 accounts).

The results show that increasing the initial balance will increase the number of committed transactions and the average committed throughput. This is easily explained: an increase in the initial balance increases the number of balance removals each account can process before conditions will fail. Hence, an increase of initial balance will causes a sharp decrease in the number of constraint failures and, as such, will cause an accompanying increase in the number of committed transactions. As constraint failures are the only reason why transactions fail to commit in protocols that use isolation-free execution, increasing the initial balance (and decreasing the likelihood of constraint failures) massively improves performance of these protocols in terms of committed transactions and average committed throughput. Furthermore, as increasing the initial balance reduces the likelihood of constraint failure, we see an accompanying increase in the number of shard-steps. This is especially the case for protocols that use linear orchestration, as these protocols all have the ability to *abort-fast* at every vote step. As parallel protocols only have limited options to abort-fast (as only the root shard can abort the entire transaction before initiating any other vote-steps), a reduction in the likelihood of constraint failure is accompanied by a more moderate increase in the number of shard-steps in these protocols.

The read-write experiment In our seventh experiment, we inspect the impact of *write-heavy transactions* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of affected accounts that are modified* by each transaction in the workload while keeping all other parameters the same (including the number of shards and the initial dataset). For each case, we generate appropriate workloads in which each transaction affects 16 accounts of which ζ accounts are modified: each such transaction will put constraints on $16 - \zeta$ accounts (read operations), remove balance from $\zeta/2$ accounts (write operations), and add balance to $\zeta/2$ accounts (write opera-

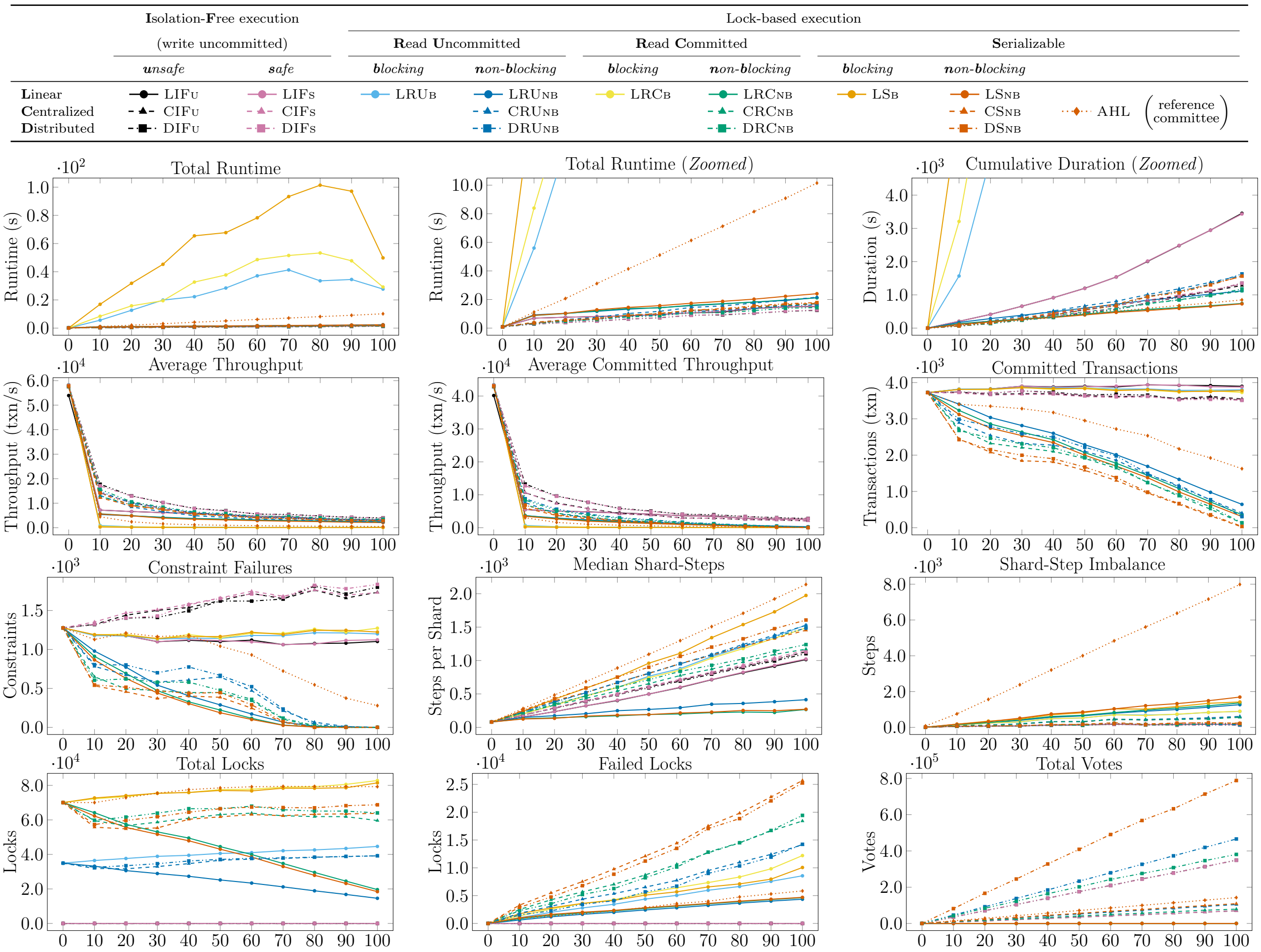

Figure 10 Measurements for the *local-global experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *percentage of global (multi-shard) transactions*. We used a fixed initial dataset with 8192 active accounts, 64 shards, and a workload of 5000 transactions (each affecting 16 accounts).

tions). The results of this experiment can be found in Figure 12.

The results show the impact of reads and writes on the lock-based protocols: we see that an increase in the number of accounts written to also increases the overall costs of all protocols that utilize lock-based execution. This is as expected, as an unlimited number of transactions can hold a read lock while only one transaction can hold a write lock on an account (during which no other transactions can obtain locks on that account), write locks are more costly than read locks and are more prone to causing lock failures.

At the same time, we see that write-heavy workloads have a secondary negative impact on performance: an increase of the number of accounts that are written to also increases the likelihood of accounts to have their balance decreased. Hence, increasing the number of accounts that are written to increases the likelihood of constraint failures and, consequently, decreases the number of committed transactions and the average committed throughput.

The transaction size experiment In our eighth and final experiment, we study the impact of the *transaction size* on the behavior of ByShard. To do so, we measured the behavior of the system as a function of the *number of accounts affected* by each transaction in the workload while keeping all other parameters the same (including the number of shards and the initial dataset). For each case, we generate appropriate workloads: in the workload in which each transaction affects ζ accounts, each such transaction will put constraints on $\zeta/2$ accounts (read operations), remove balance from $\zeta/4$ accounts (write operations), and add balance to $\zeta/4$ accounts (write operations). Notice that increasing the number of affected accounts also can increase the number of affected shards per transaction. The results of this experiment can be found in Figure 13.

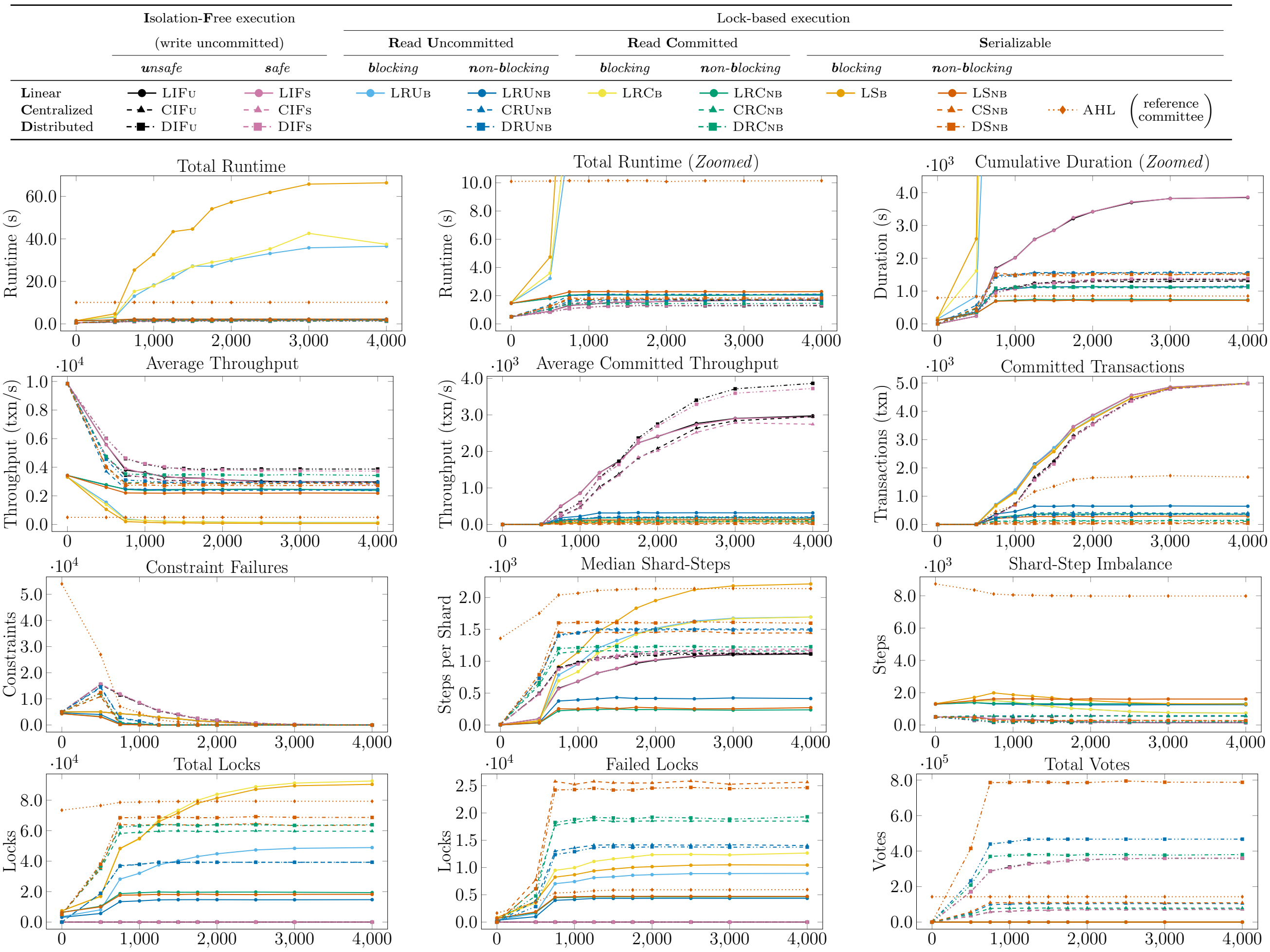

Figure 11 Measurements for the *constraint failure experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *initial balance*. We used 64 shards, 8192 accounts, and a fixed workload of 5000 transactions (each affecting 16 accounts).

The results show the obvious impact of transaction size on performance: increasing the transaction size increases the overall cost of all ByShard protocols in terms of runtime, duration, steps, shard-steps, locks, and votes. Consequently, increasing the size of transactions decreases the average throughput and the average committed throughput. Furthermore, increasing the transaction size increases the imbalance in shard-steps for the protocols that use linear orchestration. As explained in the scalability experiment, this is a side-effect of the deterministic order in which vote-steps are performed in the protocols that use linear orchestration, and this side-effect cannot be avoided for protocols that use blocking locks (without further measures to prevent deadlocks).

We also see that the protocols that use blocking lock are most impacted by an increase in transaction size, as not only the overall amount of shard-steps per transactions increases, but also the wait times for each failed lock. Furthermore, an increase in the transaction size increases the number writes and, similar to the previous experiment, increases the number of constraint failures and decreases the number of committed transactions and the average committed throughput.

7 Discussion and related work

There is abundant literature on the design and implementation of distributed systems, distributed databases, and sharding (e.g., [42,48,47]). Furthermore, there is also abundant literature on resilient systems and consensus (e.g., [6,7,10,12,38,54]). Next, we shall focus on the design decisions made by ByShard and compare them to the few works that deal with sharding in resilient systems.

Workloads in resilient systems We used a rather simple *account-transfer* data and transaction model through-

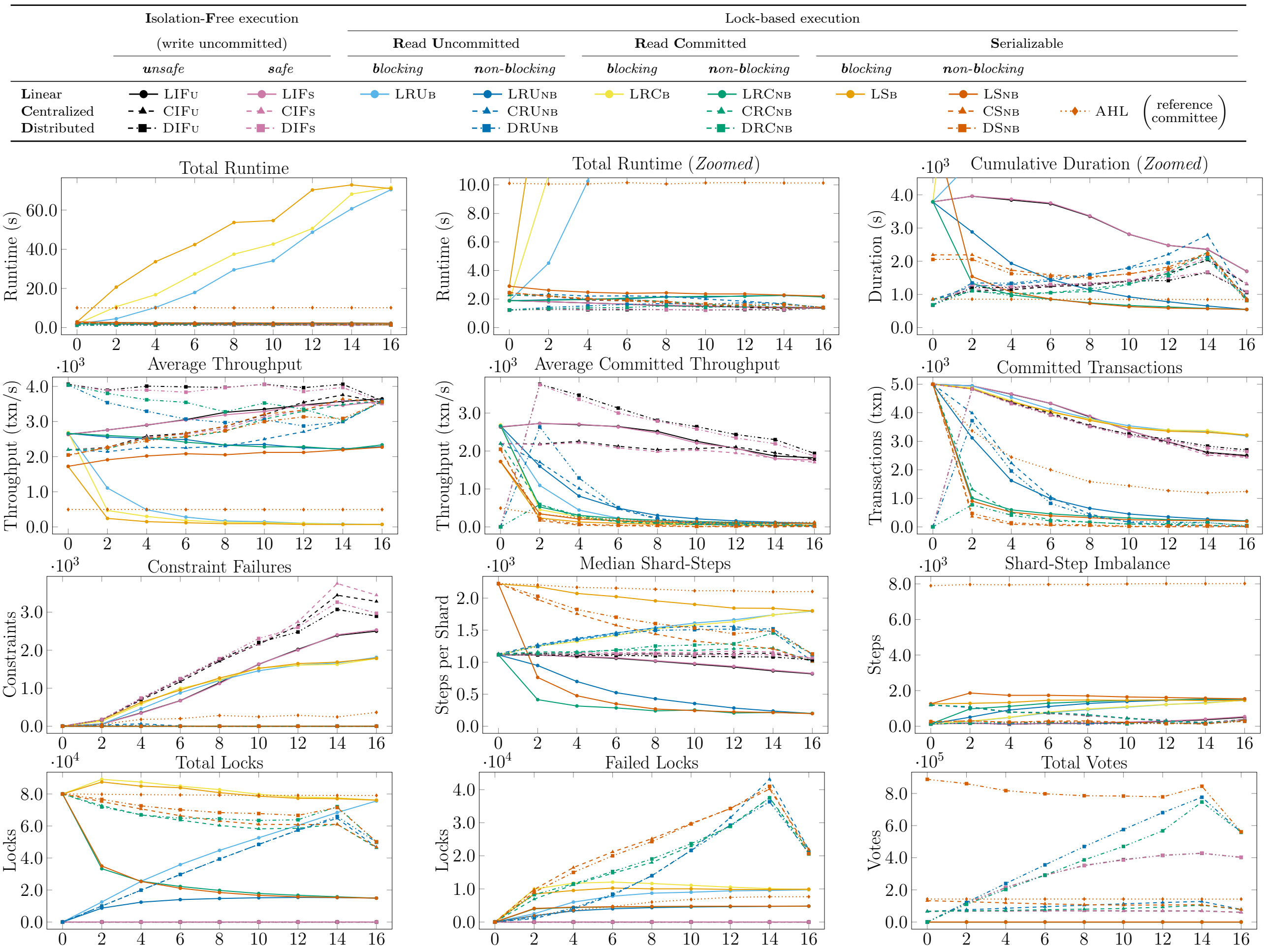

Figure 12 Measurements for the *read-write experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *number of affected accounts that are modified*. We used a fixed initial dataset with 8192 active accounts, 64 shards, and a workload of 5000 transactions (each affecting 16 accounts).

out this paper. Still, all principles outlined in this paper can be applied to *any* data and transaction model in which transactions are *one-shot transactions*: for more complex transactions, the data necessary for execution can be exchanged between the affected shards alongside votes. The main limitation of this approach is the *total amount of all data*, which depends highly on the type of workloads. This approach is not optimal for all workloads, however. Take, for example, evaluating complex joins between tables held in distinct shards, which will require huge data transfers. We believe that developing efficient sharded query evaluation algorithms for permissioned blockchains with low costs in terms of resilient primitives and exchanged data is a major direction for future work.

We have not considered *interactive transactions* that require back-and-forth steps by clients and the system. Although such interactive transactions are supported by some traditional data management systems, we believe that they are ill-suited for resilient systems: as illustrated in Example 1, interactive transaction processing in resilient systems would be costly and unresponsive due to the high cost of the individual consensus steps required to process each back-and-forth step.

The general-purpose data and transaction model used by BYSHARD is in contrast with the more restrictive UTXO-based data models that CHAINSPACE [1] and CERBERUS [25] utilize to their advantage to provide consistent transaction execution when dealing with contention and Byzantine behavior.

ByShard and decentralized sharding The design of the eighteen multi-transaction protocols of BYSHARD are *decentralized*: there is no central coordinator that is assigned the task to coordinate execution of all multi-shard transactions. This is in contrast with systems such as AHL [11] that use a reference committee to coordinate execution of all multi-shard transactions. This difference between BYSHARD and AHL is not fundamental, however, as the multi-shard transaction

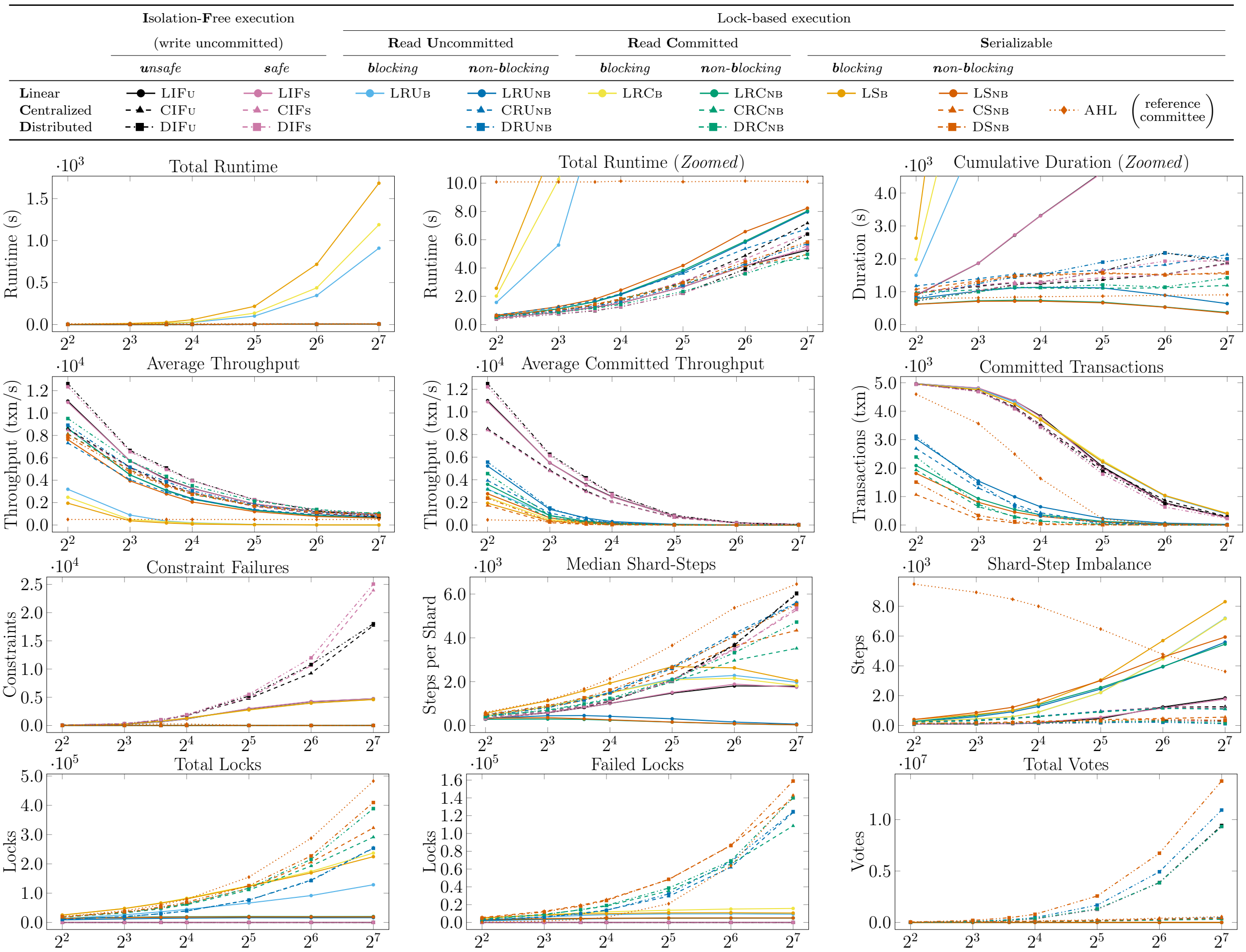

Figure 13 Measurements for the *transaction size experiment* using *nineteen* multi-shard transaction processing protocols. In this experiment, we measured the behavior of the system as a function of the *percentage of global (multi-shard) transactions*. We used a fixed initial dataset with 8192 active accounts, 64 shards, and a workload of 5000 transactions (each affecting 16 accounts).

protocol of AHL can easily be expressed within the orchestrate-execute model of ByShard.

As shown in Section 6, the usage of a central coordinator (e.g., reference committees) can significantly reduce contention while providing good scalability for single-shard workloads. At the same time, the usage of a central coordinator introduces bottlenecks when processing workloads with many multi-shard transactions.

The usage of Byzantine primitives To maximize the throughput of a sharded resilient system, we have to assure that standard performance-enhancing techniques can be applied at the single-shard level. This is especially true for *out-of-order processing* [9, 11, 19], which can increase consensus throughput in consensus-based systems by several orders of magnitudes (see Example 1). In ByShard, we assured that such performance-enhancing techniques are easily applicable by utilizing standard Byzantine primitives as basic building blocks.

This is in contrast with recent systems such as Caper [2] and SharPer [3] that minimize the *duration* of multi-shard transaction processing. To do so, these systems process each multi-shard transactions via a *single* transaction-specific multi-shard-aware consensus step, which reduces the number of consecutive consensus steps to an absolute minimum. However, such designs have difficulties dealing with contention, while making it non-trivial to apply standard performance-enhancing techniques such as out-of-order processing.

As ByShard relies on standard Byzantine primitives, the design of ByShard is highly flexible and can easily be tuned towards specific applications, e.g., by providing only crash-fault tolerance by using the Paxos consensus protocol, by minimizing communication costs by using the HotStuff consensus protocol, and so on.

Sharding in permissionless blockchains In parallel to the development of traditional resilient systems, there has been promising work on cross-blockchain coordination and sharding in permissionless blockchains such as BITCOIN [39] and ETHEREUM [51]. Examples include techniques that enable reliable cross-chain coordination via sidechains, blockchain relays, atomic swaps, atomic commitment, cross-chain deals, and distributed hash-tables [15, 16, 30, 36, 52, 56, 32, 23, 24, 29]. Unfortunately, permissionless blockchains remain several orders of magnitudes slower than comparable techniques for traditional resilient systems, due to which these permissionless blockchains remain unsuitable for high-performance data management systems.

As permissionless blockchains can provide both *consensus* (e.g., using incentive-based consensus protocols such as Proof-of-Work and Proof-of-Stakes) and *cluster-sending* (e.g., built on top of techniques that enable reliable cross-chain coordination), one can apply the design of BYSHARD to a permissionless setting. By doing so, one would obtain a sharded permissionless blockchain system with flexible multi-shard transaction processing capabilities. We note, however, that it is not evident to reach high performance with such a permissionless BYSHARD using current permissionless techniques.

8 Conclusion

In this paper, we introduced the BYSHARD framework for general-purpose sharded resilient data management systems. Additionally, we introduced the *orchestrate-execute model* (OEM) for processing multi-shard transactions in BYSHARD. Next, we showed that OEM can incorporate the necessary commit, locking, and execution steps required for processing multi-shard transactions in at-most two consensus steps per involved shard. Furthermore, we showed that common multi-shard transaction processing based on two-phase commit protocols and two-phase locking can be expressed efficiently in OEM.

Our flexible design allows for several distinct approaches towards multi-shard transaction processing, each striking its own trade-off between *throughput*, *isolation level*, *latency*, and *abort rate*. To illustrate this, we performed an in-depth comparison of the *eighteen* multi-shard transaction processing protocols of the BYSHARD framework. Our results show that each protocol supports high transaction throughput and provides scalability. Hence, we believe that the BYSHARD framework is a promising step towards flexible general-purpose ACID-compliant scalable resilient multi-shard data and transaction processing capabilities.

References

1. Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G.: Chainspace: A sharded smart contracts platform (2017). URL http://arxiv.org/abs/1708.03778
2. Amiri, M.J., Agrawal, D., Abbadi, A.E.: CAPER: A cross-application permissioned blockchain. Proc. VLDB Endow. **12**(11), 1385–1398 (2019). DOI 10.14778/3342263.3342275
3. Amiri, M.J., Agrawal, D., El Abbadi, A.: SharPer: Sharding permissioned blockchains over network clusters. In: Proceedings of the 2021 International Conference on Management of Data, pp. 76–88. ACM (2021). DOI 10.1145/3448016.3452807
4. Atluri, V., Bertino, E., Jajodia, S.: A theoretical formulation for degrees of isolation in databases. Inform. Software Tech. **39**(1), 47–53 (1997). DOI 10.1016/0950-5849(96)01109-3
5. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. SIGMOD Rec. **24**(2), 1–10 (1995). DOI 10.1145/568271.223785
6. Berger, C., Reiser, H.P.: Scaling byzantine consensus: A broad analysis. In: Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, pp. 13–18. ACM (2018). DOI 10.1145/3284764.3284767
7. Cachin, C., Vukolic, M.: Blockchain consensus protocols in the wild (keynote talk). In: 31st International Symposium on Distributed Computing, vol. 91, pp. 1:1–1:16. Schloss Dagstuhl (2017). DOI 10.4230/LIPIcs.DISC.2017.1
8. Casey, M., Crane, J., Gensler, G., Johnson, S., Narula, N.: The impact of blockchain technology on finance: A catalyst for change. Tech. rep., International Center for Monetary and Banking Studies (2018). URL https://www.cimb.ch/uploads/1/1/5/4/115414161/geneva21_1.pdf
9. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002). DOI 10.1145/571637.571640
10. Correia, M., Veronese, G.S., Neves, N.F., Verissimo, P.: Byzantine consensus in asynchronous message-passing systems: A survey. Int. J. Crit. Comput.-Based Syst. **2**(2), 141–161 (2011)
11. Dang, H., Dinh, T.T.A., Loghin, D., Chang, E.C., Lin, Q., Ooi, B.C.: Towards scaling blockchain systems via sharding. In: Proceedings of the 2019 International Conference on Management of Data, pp. 123–140. ACM (2019). DOI 10.1145/3299869.3319889
12. Dinh, T.T.A., Liu, R., Zhang, M., Chen, G., Ooi, B.C., Wang, J.: Untangling blockchain: A data processing view of blockchain systems. Trans. Knowl. Data Eng. **30**(7), 1366–1385 (2018). DOI 10.1109/TKDE.2017.2781227
13. Dolev, D.: Unanimity in an unknown and unreliable environment. In: 22nd Annual Symposium on Foundations of Computer Science, pp. 159–168. IEEE (1981). DOI 10.1109/SFCS.1981.53
14. Dolev, D.: The byzantine generals strike again. J. Algorithms **3**(1), 14–30 (1982). DOI 10.1016/0196-6774(82)90004-9
15. El-Hindi, M., Binnig, C., Arasu, A., Kossmann, D., Ramamurthy, R.: BlockchainDB: A shared database on blockchains. Proc. VLDB Endow. **12**(11), 1597–1609 (2019). DOI 10.14778/3342263.3342636
16. Ethereum Foundation: BTC Relay: A bridge between the bitcoin blockchain & ethereum smart contracts (2017). URL http://btcrelay.org
17. Gordon, W.J., Catalini, C.: Blockchain technology for healthcare: Facilitating the transition to patient-driven interoperability. Comput. Struct. Biotechnol. J. **16**, 224–230 (2018). DOI 10.1016/j.csbj.2018.06.003
18. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course, pp. 393–481. Springer-Verlag (1978). DOI 10.1007/3-540-08755-9_9

19. Gupta, S., Hellings, J., Sadoghi, M.: Fault-Tolerant Distributed Transactions on Blockchain. Synthesis Lectures on Data Management. Morgan & Claypool (2021). DOI 10.2200/S01068ED1V01Y202012DTM065
20. Gupta, S., Rahnama, S., Hellings, J., Sadoghi, M.: ResilientDB: Global scale resilient blockchain fabric. Proc. VLDB Endow. **13**(6), 868–883 (2020). DOI 10.14778/3380750.3380757
21. Gupta, S., Rahnama, S., Sadoghi, M.: Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pp. 754–764. IEEE (2020). DOI 10.1109/ICDCS47774.2020.00012
22. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983). DOI 10.1145/289.291
23. Hassanzadeh-Nazarabadi, Y., Küpçü, A., Özkasap, Ö.: LightChain: Scalable DHT-based blockchain. Parallel Distrib. Syst. **32**(10), 2582–2593 (2021). DOI 10.1109/TPDS.2021.3071176
24. Hassanzadeh-Nazarabadi, Y., Taheri-Boshrooyeh, S.: A consensus protocol with deterministic finality. In: INFOCOM 2021–IEEE Conference on Computer Communications Workshops, pp. 1–2 (2021). DOI 10.1109/INFOCOMWKSHPS51825.2021.9484527
25. Hellings, J., Hughes, D.P., Primero, J., Sadoghi, M.: Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing (2020). URL https://arxiv.org/abs/2008.04450
26. Hellings, J., Sadoghi, M.: Byshard: Sharding in a byzantine environment. Proc. VLDB Endow. **14**(11), 2230–2243 (2021). DOI 10.14778/3476249.3476275
27. Hellings, J., Sadoghi, M.: Byzantine cluster-sending in expected constant communication (2021). URL https://arxiv.org/abs/2108.08541
28. Hellings, J., Sadoghi, M.: The fault-tolerant cluster-sending problem. In: Foundations of Information and Knowledge Systems, pp. 168–186. Springer (2022). DOI 10.1007/978-3-031-11321-5_10
29. Hentschel, A., Hassanzadeh-Nazarabadi, Y., Seraj, R., Shirley, D., Lafrance, L.: Flow: Separating consensus and compute–block formation and execution (2020). URL https://arxiv.org/abs/2002.07403
30. Herlihy, M.: Atomic cross-chain swaps. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, pp. 245–254. ACM (2018). DOI 10.1145/3212734.3212736
31. Herlihy, M.: Blockchains from a distributed computing perspective. Commun. ACM **62**(2), 78–85 (2019). DOI 10.1145/3209623
32. Herlihy, M., Liskov, B., Shrira, L.: Cross-chain deals and adversarial commerce. The VLDB Journal (2021). DOI 10.1007/s00778-021-00686-1
33. Herzog, T.N., Scheuren, F.J., Winkler, W.E.: Data Quality and Record Linkage Techniques. Springer (2007). DOI 10.1007/0-387-69505-2
34. Kamel Boulos, M.N., Wilson, J.T., Clauson, K.A.: Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. Int. J. Health. Geogr **17**(1), 1211–1220 (2018). DOI 10.1186/s12942-018-0144-x
35. Kamilaris, A., Fonts, A., Prenafeta-Boldύ, F.X.: The rise of blockchain technology in agriculture and food supply chains. Trends in Food Science & Technology **91**, 640–652 (2019). DOI 10.1016/j.tifs.2019.07.034
36. Kwon, J., Buchman, E.: Cosmos whitepaper: A network of distributed ledgers (2019). URL https://cosmos.network/cosmos-whitepaper.pdf
37. Lamport, L.: Paxos made simple. ACM SIGACT News **32**(4), 51–58 (2001). DOI 10.1145/568425.568433. Distributed Computing Column 5
38. Lao, L., Li, Z., Hou, S., Xiao, B., Guo, S., Yang, Y.: A survey of IoT applications in blockchain systems: Architecture, consensus, and traffic modeling. ACM Comput. Surv. **53**(1) (2020). DOI 10.1145/3372136
39. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. URL https://bitcoin.org/en/bitcoin-paper
40. Narayanan, A., Clark, J.: Bitcoin's academic pedigree. Commun. ACM **60**(12), 36–45 (2017). DOI 10.1145/3132259
41. Nathan, S., Govindarajan, C., Saraf, A., Sethi, M., Jayachandran, P.: Blockchain meets database: Design and implementation of a blockchain relational database. Proc. VLDB Endow. **12**(11), 1539–1552 (2019). DOI 10.14778/3342263.3342632
42. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Springer (2020). DOI 10.1007/978-3-030-26253-2
43. Pisa, M., Juden, M.: Blockchain and economic development: Hype vs. reality. Tech. rep., Center for Global Development (2017). URL https://www.cgdev.org/publication/blockchain-and-economic-development-hype-vs-reality
44. Reinsel, D., Gantz, J., Rydning, J.: Data age 2025: The digitization of the world, from edge to core. Tech. rep., IDC (2018). URL https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf
45. Rejeb, A., Keogh, J.G., Zailani, S., Treiblmaier, H., Rejeb, K.: Blockchain technology in the food industry: A review of potentials, challenges and future research directions. Logistics **4**(4) (2020). DOI 10.3390/logistics4040027
46. Skeen, D.: A quorum-based commit protocol. Tech. rep., Cornell University (1982)
47. van Steen, M., Tanenbaum, A.S.: Distributed Systems, 3th edn. Maarten van Steen (2017). URL https://www.distributed-systems.net/
48. Tel, G.: Introduction to Distributed Algorithms, 2nd edn. Cambridge University Press (2001)
49. The Hyperledger White Paper Working Group: An introduction to Hyperledger. Tech. rep., The Linux Foundation (2018)
50. Treiblmaier, H., Beck, R. (eds.): Business Transformation through Blockchain. Springer (2019). DOI 10.1007/978-3-319-98911-2
51. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. URL https://gavwood.com/paper.pdf. EIP-150 revision
52. Wood, G.: Polkadot: vision for a heterogeneous multi-chain framework (2016). URL https://polkadot.network/PolkaDotPaper.pdf
53. Wu, M., Wang, K., Cai, X., Guo, S., Guo, M., Rong, C.: A comprehensive survey of blockchain: From theory to IoT applications and beyond. Internet Things J **6**(5), 8114–8154 (2019). DOI 10.1109/JIOT.2019.2922538
54. Xiao, Y., Zhang, N., Lou, W., Hou, Y.T.: A survey of distributed consensus protocols for blockchain networks. Commun. Surv. Tutor **22**(2), 1432–1465 (2020). DOI 10.1109/COMST.2020.2969706
55. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 347–356. ACM (2019). DOI 10.1145/3293611.3331591
56. Zakhary, V., Agrawal, D., El Abbadi, A.: Atomic commitment across blockchains. Proc. VLDB Endow. **13**(9), 1319–1331 (2020). DOI 10.14778/3397230.3397231