Citation in BibTeX format

RESEARCH-ARTICLE

Adaptive Sharding in Untrusted Environments

BHAVANA MEHTA, University of Pennsylvania, Philadelphia, PA, United States

NUPUR BAGHEL, University of Pennsylvania, Philadelphia, PA, United States

MOHAMMAD JAVAD AMIRI, Stony Brook University, Stony Brook, NY, United States

BOON THAU LOO, University of Pennsylvania, Philadelphia, PA, United States

RYAN MARCUS, University of Pennsylvania, Philadelphia, PA, United States

Adaptive Sharding in Untrusted Environments

BHAVANA MEHTA, University of Pennsylvania, USA
NUPUR BAGHEL, University of Pennsylvania, USA
MOHAMMAD JAVAD AMIRI, Stony Brook University, USA
BOON THAU LOO, University of Pennsylvania, USA
RYAN MARCUS, University of Pennsylvania, USA

Distributed data management systems employ data sharding techniques to achieve scalability. Traditional sharding approaches typically operate under the assumption of a trusted environment, where nodes may crash, but do not act adversarially. In untrustworthy environments, however, this assumption is no longer valid. This paper presents MARLIN, an adaptive scalable data management system specifically designed for untrustworthy environments. MARLIN leverages data sharding to enhance scalability while dynamically redistributing data across clusters to adapt to dynamic workloads. We propose two architectures: a centralized architecture serving as a baseline, which employs hypergraph partitioning within a trusted administrative domain, and a decentralized architecture that eliminates the need for such a trusted domain by managing shards across nodes in a decentralized manner. Both architectures utilize real-time monitoring and adaptive algorithms to dynamically adjust sharding in response to workload characteristics and adversarial conditions. Experimental results show that MARLIN maintains consistent performance under diverse dynamic scenarios in untrustworthy environments by continuously optimizing shard distributions.

CCS Concepts: • **Information systems** → **Data management systems**; Distributed database transactions; • **Security and privacy** → *Distributed systems security*.

Additional Key Words and Phrases: Byzantine Fault Tolerance, Scalability, Adaptivity, Dynamic Sharding

1 Introduction

Traditionally, data management systems have been operated under the control of central authorities, typically assuming a trusted environment and only accommodating crash failures, where the system stops working without exhibiting any unpredictable behavior. However, incidents such as the recent collapse of multinational banks [82], the unexpected deactivation of user accounts [21], and glitches in major stock exchanges [70] have deteriorated trust in these centralized systems. As a result,

there has been a rapid emergence of decentralized systems, where numerous untrusting parties collaborate on managing data.

This shift has made untrusted environments more common and has shifted the focus of fault tolerance towards managing Byzantine failures, where nodes may exhibit arbitrary and potentially malicious behavior. With this paradigm shift, a wide range of distributed and decentralized applications has flourished recently, encompassing real-world asset tokenization [76], gaming [3], supply chain assurance [11, 83], crowdworking [9], central bank digital currencies [1], and decentralized AI inference [2].

Distributed data management rely on fault-tolerant protocols to provide robustness and high availability [19, 24, 26, 31, 34, 48, 62]. While trusted cloud systems [26, 31, 34], rely on crash fault-tolerant (CFT) protocols, e.g., Paxos [55], to establish consensus, systems deployed in untrustworthy environments use Byzantine fault-tolerant (BFT) protocols, e.g., PBFT [28], to deal with adversarial behaviors. Some notable examples of such systems are blockchains [6, 10, 13, 20, 27, 53, 68], lock servers [29], certificate authority systems [92], firewalls [23, 39, 40, 71, 79, 88], SCADA systems [18, 49, 65, 91], key-value datastores [22, 36, 43, 46, 71], and key management [60].

As businesses and applications produce evergrowing volumes of data, data management systems need to scale to manage this growth efficiently and support more users while maintaining a consistent performance. Partitioning the data into multiple shards maintained by different clusters of trusted nodes is a proven approach to enhance the scalability of distributed databases [19, 26, 31, 34, 42, 48, 80, 81]. A key challenge in scalable data management is developing an optimal data sharding strategy that efficiently allocates data items to various clusters of nodes while balancing competing objectives, such as load balancing, minimizing cross-shard transactions, and optimizing throughput and latency. While traditional partitioning techniques, e.g., round-robin, range-based, and hash-based partitioning [35], exist, they often lead to a significant number of cross-shard transactions. Cross-shard transactions access data items across multiple shards, and processing such transactions requires running atomic commitment protocols, e.g., two-phase commit [57], across clusters, layered atop consensus protocols, resulting in unacceptable latencies [47]. What was optimal yesterday may not be optimal today. As workloads evolve, new data is added, or users migrate, previously optimal sharding schemes can become arbitrarily poor. To overcome the limitations of fixed, workload-oblivious data sharding, researchers have proposed dynamic workload-aware partitioning strategies, such as hypergraph partitioning [32, 69]. In these schemes, data items are modeled as vertices and transactions as hyperedges, allowing for partitioning that minimizes edge cuts while keeping each segment sufficiently small to fit within a single node.

Unfortunately, current dynamic sharding systems [32, 69, 74] are primarily designed for trusted environments and are unsuitable for Byzantine environments. For example, consider the incremental repartitioning strategy proposed in [69], which moves data between shards when the percentage of cross-shard transactions crosses a threshold. An adversary could trigger near-constant repartitioning by issuing queries that touch a random set of tuples, and are therefore highly likely to cross partition boundaries. Such an adversary could quickly force the system into unwanted repartitioning, severely degrading the performance of legitimate clients. Although data sharding under Byzantine failures has been widely studied [12, 64], especially in the domain of blockchains [7, 8, 10, 12, 14, 33, 38, 41, 50, 59, 90], most existing systems rely on static, workload-oblivious sharding and do not adaptively reshard data in response to changes in the workload.

The need for adaptive sharding in untrusted environments arises from different real-world applications where traditional trusted solutions fail to meet their requirements [61]. In particular, emerging multi-enterprise applications, e.g., decentralized supply chain management [6, 51] and multi-platform crowdworking [9], consist of mutually distrustful enterprises that collaboratively process data while maintaining competitive advantages [11]. For example, a supply chain application

may experience highly variable workloads based on seasonal demand, product launches, and supply disruptions. A static partitioning scheme optimal for normal operations becomes inefficient during peak seasons or supply chain disruptions, requiring dynamic adaptation without trusting any single party to coordinate resharding decisions. As another example, cross-border financial systems increasingly require cross-entity collaboration without centralized control, e.g., central bank digital currencies (CBDCs) [1] involve multiple national banks that cannot trust each other with a centralized coordinator, yet must handle varying transaction volumes across different time zones and economic events. Adaptive sharding would also benefit many other applications deployed in untrusted environments, such as decentralized identity and reputation systems [15] and healthcare [17], where there is a need to adapt to varying regional demand, network conditions, and resource availability.

Untrustworthy environments present several unique challenges to the adaptive data sharding problem, as BFT protocols are inherently more complex and resource-intensive than CFT protocols. In particular, CFT protocols, e.g., Paxos [55], require $2f + 1$ nodes to overcome the simultaneous crash failure of any f nodes and establish consensus with linear message complexity. BFT protocols, e.g., PBFT [28], on the other hand, require $3f+1$ nodes and typically incur quadratic communication complexity to reach agreement on the order of transactions. This gap becomes more exaggerated when processing cross-shard transactions, which necessitate running atomic commitment protocols on top of consensus protocols in each involved cluster. The absence of a trusted coordinator in untrusted settings results in elevated communication costs, as coordination must rely on a Byzantine fault-tolerant cluster of nodes to assume the coordinator's role [10, 33], or fully decentralized coordination approaches [8]. Additionally, BFT protocols require advanced mechanisms to verify the authenticity of messages. These challenges necessitate the design of an adaptive system that minimizes cross-shard transactions, thereby reducing the coordination burden associated with processing such transactions.

This paper introduces Marlin, a scalable distributed data management system designed to tolerate Byzantine faults while efficiently adapting to dynamic workloads. Marlin introduces adaptive sharding mechanisms that dynamically refine data shards based on real-time workload characteristics and minimize cross-shard communication in untrustworthy environments. Marlin relies on BFT consensus protocols to establish agreement within each cluster while employing a coordinator-based BFT atomic commitment protocol to process cross-shard transactions.

The main contribution of Marlin lies in integrating Byzantine fault-tolerant consensus protocols, atomic commitment protocols, and dynamic sharding algorithms in untrusted environments. This integration, however, introduces several unique technical challenges.

First, resharding decisions in dynamic trusted environments can be made by a single agent. However, the absence of a trusted party in untrusted settings necessitates consensus among multiple nodes for each resharding decision. This shifts the problem from local optimization to distributed agreement, resulting in significantly higher communication complexity. Traditional dynamic sharding algorithms rely on a global perspective for optimal decisions; in contrast, our system must enable nodes to collaboratively converge on effective sharding decisions using only the local information of different nodes. This fundamentally changes the cost-benefit analysis of when and how to reshard.

Second, resharding decisions depend on accurate performance metrics; however, Byzantine nodes may report misleading statistics to either trigger unnecessary resharding or prevent beneficial resharding. The system must effectively distinguish between legitimate performance degradation and adversarial manipulation while being responsive to genuine workload variations.

Finally, once resharding decisions are made, atomic commitment protocols are needed to move data across different shards. Unlike the traditional atomic commitment protocol (e.g., two-phase

commit), where a single trusted entity coordinates the process, our system must ensure that resharding proposals and their execution remain atomic even when both the proposing and receiving shards contain malicious nodes attempting to disrupt the process.

Marlin presents centralized and decentralized architectures to tackle these challenges. The centralized architecture, which serves as a baseline, relies on a trusted administrative domain for efficient shard management, uses hypergraph partitioning to make resharding decisions, and integrates PBFT with the 2PC atomic commitment protocol to perform resharding. The decentralized architecture, on the other hand, eliminates reliance on a central trusted component using a key affinity resharding algorithm where nodes collaboratively determine shard assignments in a decentralized manner.

We conduct extensive experiments under various workloads and fault conditions. Our results demonstrate that Marlin effectively adapts to dynamic workloads and maintains consistent throughput even when Byzantine failures occur. The centralized architecture attains faster convergence and higher throughput due to coordinated shard management, while the decentralized architecture offers enhanced resilience by eliminating the need for trusted components.

2 Background

Sharding is a widely adopted technique for improving scalability in distributed systems [31, 34]. By partitioning data across multiple fault-tolerant clusters of nodes, systems manage large-scale data efficiently. Sharding strategies are generally categorized into *static* and *dynamic*.

Static sharding strategies partition data into fixed shards using a predefined scheme, e.g., hash-based partitioning, round-robin partitioning, and range partitioning. Implementing static sharding to distribute data across nodes has been used in several data management systems, e.g., MySQL Cluster [31] and MongoDB [58]. This method is effective for stable workloads, as it minimizes coordination overhead and avoids the complexity associated with frequent data resharding. However, static sharding can lead to load imbalances and increased cross-shard communication when workloads shift, requiring manual intervention for data redistribution.

In contrast, dynamic sharding adjusts shard boundaries in response to changing workloads, optimizing load balance and minimizing cross-shard transactions. Systems such as SWORD [69], Schism [32], Spanner [31], and CosmosDB [38] implement dynamic sharding to adaptively reshard data based on real-time workload statistics.

Most existing sharding solutions, especially those employing dynamic sharding [32, 69], assume a *trusted environment*, where the failure model of nodes is fail-stop. In the fail-stop failure model, nodes operate at arbitrary speed, may fail by stopping, and may restart; however, they may not collude, lie, or otherwise attempt to subvert the protocol. Crash fault-tolerant protocols, e.g., Paxos [55], guarantee safety in an asynchronous network using $2f+1$ nodes to overcome the simultaneous crash failure of any f nodes. To process cross-shard transactions in a trusted environment, atomic commitment protocols, e.g., two-phase commit and three-phase commit, have been used. In trusted environments, the primary focus is on performance and scalability, without the necessity to deal with adversarial threats. This assumption simplifies the design of sharding mechanisms but overlooks the challenges posed by untrusted environments where nodes may behave maliciously.

Untrusted environments, on the other hand, follow the Byzantine failure model where faulty nodes may exhibit arbitrary, potentially malicious, behavior [56]. Ensuring system reliability and consistency in such scenarios requires implementing Byzantine Fault Tolerance (BFT) protocols, e.g., PBFT [28] and HotStuff [89]. BFT protocols need $3f+1$ nodes to provide safety in the presence of f malicious nodes [25]. Sharding in untrusted environments is more complex due to the need for communication and coordination across untrustworthy shards.

	Static	Dynamic
Trusted	**Data Warehouses [34]**	**SWORD [69]**
Untrusted	**SharPer [8]**	**Marlin**

Fig. 1. Classification of sharding strategies based on trust assumptions and sharding dynamics

2.1 Static Sharding in Untrusted Contexts

Systems deployed in untrustworthy environments mainly rely on BFT protocols to establish consensus on the order of transactions. Most distributed data management systems deployed in untrusted environments opt for *static sharding* to reduce coordination overhead and limit the attack surface where a malicious node attempts to manipulate resharding decisions [8, 78].

Distributed data management systems need to process *intra-shard* and *cross-shard* transactions. While intra-shard transactions are processed locally within a cluster using a BFT consensus protocol, cross-shard transactions require coordination across all clusters that maintain corresponding data shards. Cross-shard transactions can be processed in a centralized coordinator-based manner, e.g., AHL [33], Saguaro [10], and Blockplane [64], or in a decentralized flattened way, e.g., SharPer [8].

We focus on the coordinator-based approach, where an atomic commitment protocol, e.g., two-phase commit (2PC), is used to process cross-shard transactions across shards. In contrast to the implementations of the 2PC protocol in trusted environments, where a single coordinator manages the commitment of each cross-shard transaction, in untrustworthy environments, the coordinator itself needs to tolerate Byzantine failures.

As a result, committing a cross-shard transaction in untrustworthy environments becomes much more expensive, involving three rounds of cross-cluster communication (i.e., prepare, prepared and commit) and two instances of Byzantine Fault Tolerant (BFT) consensus (one for prepare phase and one for commit phase) within each cluster, each requiring multiple phases of all-to-all communication. Consequently, minimizing the percentage of cross-shard transactions in these systems could lead to a substantial improvement in overall performance.

2.2 Dynamic Sharding in Untrusted Contexts

Existing scalable data management solutions primarily focus on either static sharding in untrustworthy environments [8, 33], which lacks adaptability, or provide dynamic sharding in trusted environments [32, 69] where there is no Byzantine failure. This gap results in the poor performance of scalable data management systems when deployed in dynamic, untrusted environments where workloads change frequently.

Employing dynamic sharding techniques in untrusted environments, however, poses significant challenges. Dynamic sharding requires frequent adjustments to shard boundaries, which, in an untrusted environment, necessitates communication and coordination across potentially malicious nodes. The additional communication overhead and complexity of achieving consensus in BFT systems make it difficult to adapt shard boundaries efficiently. Furthermore, while a trusted co-ordinator can make resharding decisions in trusted contexts, such decisions must be made in a decentralized manner in untrustworthy environments.

This paper introduces Marlin, the first distributed data management system, to our knowledge, that supports adaptive data sharding in untrusted environments where nodes may exhibit Byzantine failures. Although BFT consensus protocols, atomic commitment protocols, and dynamic sharding algorithms have been extensively studied in isolation, their integration for enabling adaptive resharding under Byzantine conditions presents unique challenges that have not been addressed in prior work. We formalize the problem of adaptive sharding in untrusted environments, where neither centralized coordination nor individual node trustworthiness can be assumed. Prior systems have addressed either static sharding in untrusted environments (e.g., AHL [33], Saguaro [10], SharPer [8]) or adaptive sharding in trusted environments (e.g., SWORD [69], Schism [32]). Our work bridges this gap by supporting dynamic sharding in untrusted environments. Figure 1 categorizes sharding strategies based on trust assumptions and sharding dynamics, highlighting the positioning of Marlin relative to existing systems. In such a setting, nodes need to collaborate, negotiate, and reach consensus to establish resharding decisions in a decentralized manner, while each node individually does not have a global view of the system. The untrustworthiness of nodes and the decentralized nature of the proposed algorithm introduce unique challenges not present in traditional scalable data management systems.

3 Marlin Model

Marlin operates in a distributed system consisting of a network of N nodes and C clusters. Each cluster C_i contains distinct $N_i = 3f_i + 1$ nodes, where $\Sigma N_i = N$ and f_i is the maximum number of faulty nodes that cluster C_i can tolerate concurrently [56]. In scalable data management systems, in order to guarantee *deterministic* safety, data shards are assigned to *pre-determined* fault-tolerant clusters, e.g., cloud environments. Conversely, some scalable blockchain systems, e.g., OmniLedger [50] and AHL [33], *configure* clusters themselves and provide *probabilistic* safety guarantees. Due to the absence of well-defined fault-tolerant clusters, these systems randomly assign nodes to clusters to achieve a uniform distribution of faulty nodes. Specifically, clusters are formed such that for each cluster C_i, with high probability, $|C_i| \geq 3f_i + 1$ where f_i denotes the number of faulty nodes within cluster C_i. To achieve safety with a high probability, e.g., $1 - 2^{-20}$, these clusters often need to be large in size, e.g., 80 nodes in AHL.

Marlin, similar to scalable data management systems, assumes the existence of pre-determined fault-tolerant clusters and provides deterministic safety guarantees. We assume cloud providers offer such clusters and guarantee their fault tolerance: no single cluster C_i contains more than f_i (one-third) Byzantine nodes; otherwise, there will be no safety guarantee, as it violates the requirements of the underlying BFT protocol. It is important to highlight that sharded systems, in general, are more vulnerable to attacks, such as denial-of-service, compared to flat systems with the same number of nodes. This is because attackers can more easily target a specific cluster instead of the entire znetwork, illustrating a trade-off between scalability and security.

In Marlin, all nodes are involved in both ordering and execution, i.e., there are no light nodes. Light nodes are used in permissionless blockchains like Bitcoin [63] for verification purposes by storing only a subset of the blockchain data, such as block headers, and do not engage in the consensus process, like Proof of Work mining. This approach is primarily due to the significant size of the state data and the high costs associated with mining. In permissioned environments, such as our system, the consensus is achieved through less computationally expensive protocols, and the state data is typically much smaller, making the use of light nodes potentially inefficient. Moreover, our decentralized architecture requires nodes to have access to local performance metrics, transaction histories, and current shard maps to make informed resharding decisions. Light nodes, with their limited state and reduced participation, would be unable to contribute meaningfully to the resharding process, which is a core function of our system.
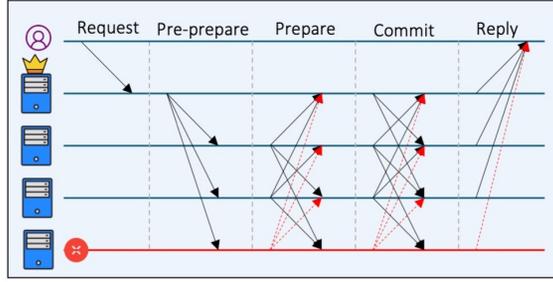
Fig. 2. Practical Byzantine Fault-Tolerant (PBFT) Protocol

Data in Marlin is partitioned into *shards*, with each shard d_i managed by a single cluster C_i. Sharding enhances scalability by distributing data and workloads across multiple clusters. Each shard is replicated across all nodes within its respective cluster. To tolerate Byzantine failures within clusters, Marlin employs Byzantine fault-tolerant protocols. BFT protocols use the State Machine Replication (SMR) algorithm [54, 73] where the system provides a replicated service whose state is mirrored across different deterministic replicas. At a high level, the goal of a BFT SMR protocol is to assign each client request an order in the global service history and execute it in that order [77]. In a BFT SMR protocol, all non-faulty replicas execute the same requests in the same order (*safety*) and all correct requests are eventually executed (*liveness*).

In an asynchronous system, where replicas can fail, no consensus solutions guarantee both safety and liveness (FLP result) [37]. Marlin, similar to most practical BFT systems, assumes the partial synchrony model, where there exists an unknown global stabilization time (GST), after which all messages between correct replicas are received within some known bound Δ.

Marlin further inherits the standard assumptions of existing BFT protocols. First, while there is no upper bound on the number of faulty clients, as mentioned before, the maximum number of concurrent malicious nodes in each cluster C_i is assumed to be f_i. Second, replicas are connected via an unreliable network that might drop, corrupt, or delay messages. Third, the network uses point-to-point bi-directional communication channels to connect replicas. Fourth, the failures of replicas are independent of each other, where a single fault does not lead to the failure of multiple replicas. Finally, a strong adversary can coordinate malicious replicas and delay communication. However, the adversary cannot subvert cryptographic assumptions.

The intra-shard consensus BFT consensus protocol is pluggable, and Marlin can be deployed with any BFT protocol. The current deployment of Marlin uses PBFT [28]. In PBFT, and during a normal case execution, a client sends a request to the leader (primary) replica. The leader assigns a sequence number to the request to determine the execution order of the request and multicasts a pre-prepare message to all other replicas (*backups*). Upon receiving a valid pre-prepare message from the leader, each backup replica multicasts a prepare message to all replicas and waits for prepare messages from $2f$ different replicas (including the replica itself) that match the pre-prepare message. The goal of the prepare phase is to guarantee safety within the view, i.e., $2f$ replicas received matching pre-prepare messages from the leader and agree with the order of the request.

Each replica then multicasts a commit message to all replicas. Once a replica receives $2f + 1$ valid commit messages from different replicas, including itself, that match the pre-prepare message, it commits the request. The goal of the commit phase is to ensure safety across views, i.e., the request is replicated on a majority of non-faulty replicas and can be recovered after (leader) failures. The second and third phases have $O(n^2)$ message complexity.
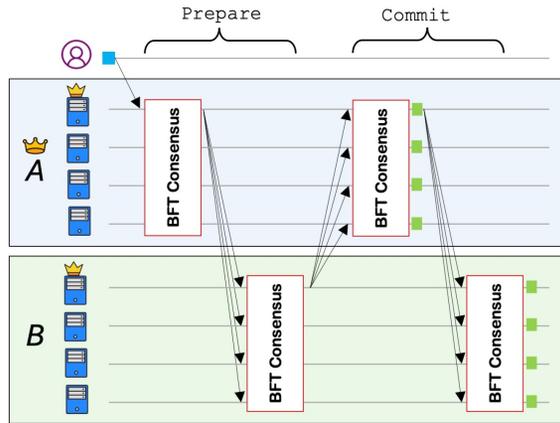
Fig. 3. Byzantine Fault-Tolerant Two-Phase Commit Protocol for Cross-Shard Transactions

If the replica has executed all requests with lower sequence numbers, it executes the request and sends a reply to the client. Finally, the client waits for $f + 1$ valid matching responses from different replicas to make sure at least one correct replica executed its request. PBFT also has a view change routine that provides liveness by allowing the system to make progress when the leader fails. Figure 2 shows the normal case operation of the PBFT protocol.

To process cross-shard transactions, Marlin relies on a Two-Phase Commit protocol integrated with BFT consensus at each phase (BFT-2PC). Figure 3 presents a coordinator-based BFT cross-shard protocol across two clusters A and B where cluster A plays the coordinator role. As can be seen, the leader of the coordinator cluster initiates the protocol by locally processing the incoming request (using a BFT protocol) and multicasting a prepare message, including the request, to all other involved clusters (i.e., B). Cluster B then independently uses a BFT consensus protocol to internally reach agreement on the order of the request and sends its vote using a prepared message to the coordinator cluster. Upon receiving prepared messages from every involved cluster (here, only cluster B), an instance of the BFT consensus protocol will be initiated again among nodes of the coordinator cluster to decide whether to commit or abort the transaction, where all other clusters become aware of the decision. Reaching a local agreement in each phase of the protocol is needed to prevent a malicious leader from sending invalid messages to other clusters.

Note that the coordinator cluster does not necessarily need to be a participant cluster and can be a distinct cluster of nodes, as illustrated in AHL [33] and Saguaro [10]. While the separation of the coordinator simplifies the 2PC-BFT logic, it introduces the following challenges: (1) increased resource costs, as a separate cluster of nodes is required to fulfill the coordinator role (without directly contributing to transaction processing capacity), which could otherwise be utilized to maintain a data shard and process transactions for enhanced performance, (2) potentially higher communication costs, which may vary depending on the placement of the coordinator and participant clusters, and (3) maintaining a separate coordinator cluster in an untrusted environment could potentially introduce additional security vulnerabilities as the cluster becomes a high-value target for attackers.

4 Centralized Architecture

The centralized architecture of Marlin serves as a baseline model for distributed databases deployed in untrusted environments.
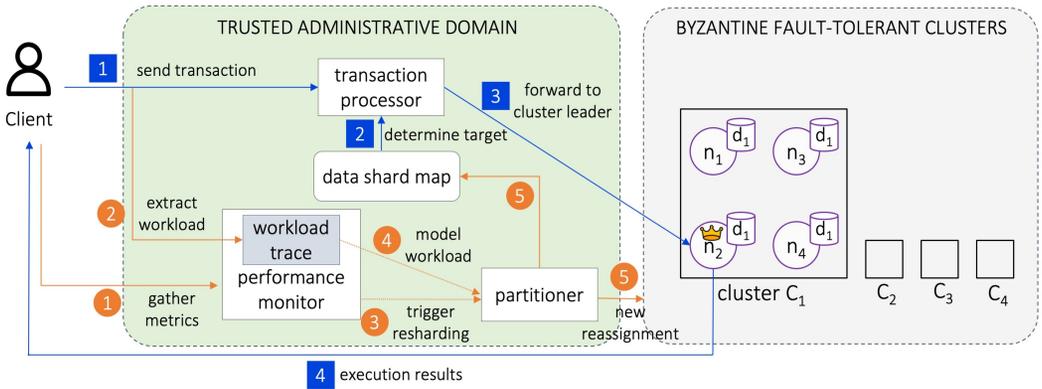
Fig. 4. Centralized Architecture

We consider a system composed of nodes organized into distinct clusters, or shards. Each cluster is assumed to correctly handle its own internal operations by tolerating up to f Byzantine failures via a standard BFT consensus protocol. The scope of our work is orthogonal to ensuring safety (i.e., data correctness), as this is guaranteed by the underlying BFT implementation within each shard. Rather, our focus is on a class of performance degradation attacks, wherein an adversary acting as a client submits a carefully crafted workload. The objective of such a workload is to induce frequent, unnecessary data migrations, thereby degrading system throughput for legitimate clients in what amounts to a Denial-of-Service (DoS) attack. The mechanisms presented in this paper are therefore designed to mitigate these specific workload-driven attacks and do not claim to address other threat vectors, such as network-level attacks or the compromise of more than f nodes within a single cluster.

As shown in Figure 4, the replicas are organized into clusters, each managed by a cluster leader (e.g., the leader node of the PBFT protocol). The clusters are, in turn, managed by a *central coordinator* that maintains a global view of the entire system. The coordinator performs operations such as transaction routing, shard allocation, maintaining the global shard map, and performance monitoring, collectively referred to as *global operations*.

While this architecture relies on a trusted centralized coordinator, it is designed to tolerate Byzantine failures within clusters. Each cluster employs the PBFT consensus protocol to ensure that intra-shard transactions are executed correctly, even in the presence of up to f faulty nodes. However, the central coordinator itself is not replicated. It is important to highlight that, in general, achieving fault tolerance through the replication of the coordinator across a cluster of nodes is not particularly difficult. However, since the centralized design is used as the baseline model in Marlin, we chose to deploy the coordinator on a single node.

When the central coordinator receives a client request (1 in Figure 4), the transaction processor determines the appropriate shard for data access. It uses the data shard map (2) to locate the shard where the data resides and forwards the transactions to the leader of the corresponding cluster (3). If more than one shard is involved in the transaction (i.e., a cross-shard transaction), the transaction is randomly forwarded to the leader of one of the involved clusters.

Upon receiving the transactions, the cluster leader (e.g., node n_2 in cluster C_1) initiates PBFT to reach an agreement on the transaction order. Depending on the transaction type, the leader either runs PBFT within its cluster for intra-shard transactions or initiates the BFT-2PC protocol for

cross-shard transactions, as described in Section 3. Once the transaction is executed, the response is sent back to the client by all nodes following the PBFT protocol (④).

4.1 Monitoring and Shard Management

The performance monitor assesses system performance by collecting metrics such as throughput and latency (① in Figure 4). It monitors both client requests and replies from clusters. Since replies may arrive at different times due to network delays or processing times, the performance monitor correlates requests and corresponding replies using unique transaction identifiers. It aggregates results over predefined time intervals to accurately measure performance even when replies are delayed. A workload trace, i.e., a historical log of transactions, is maintained to analyze performance trends.

Upon detecting performance degradation, characterized by a significant drop in throughput or a notable increase in latency, the performance monitor initiates the resharding process (②). Performance degradation is quantified based on predetermined system-specific thresholds, such as a throughput drop exceeding 40%.

The partitioner is responsible for optimizing shard distribution to balance the workload across clusters and minimize cross-shard communication. When the performance monitor triggers a resharding operation (③), the partitioner initiates the resharding process by modeling the database workload as a *hypergraph* (④). In this hypergraph, vertices represent data items (keys), and hyperedges represent transactions that access those data items. This representation captures data affinity relationships, as keys frequently accessed together in transactions are connected by hyperedges. The partitioner uses hypergraph partitioning algorithms, specifically, KaHyPar [72], to partition the hypergraph into k balanced partitions (shards) while minimizing the number of cross-shard transactions (the hyperedge cut). This optimization produces a candidate key-to-shard assignment.

Subsequently, the coordinator performs migration cost analysis by comparing the candidate sharding with the current sharding scheme. The system computes the cost of migrating each key from its current shard to its target shard, considering factors such as data size and transfer overhead. The Hungarian algorithm is employed to solve this assignment problem optimally, matching key sets to physical shards while minimizing the total migration cost. The coordinator then identifies candidate keys for migration based on a benefit-to-cost analysis. Keys are ranked according to their potential for reducing cross-shard transactions per unit of migration cost, where the benefit represents the reduction in cross-shard traffic and the cost represents the migration overhead. Finally, the coordinator selects the highest-ranked keys for migration in the current resharding round, subject to system constraints such as preventing any single shard from being overwhelmed by simultaneous incoming key migrations. The coordinator then manages the migration of keys using the BFT-2PC protocol described in Section 3. After partitioning, the data shard map is updated with the new reassignments (⑤), which are shared with the other clusters.

4.2 Hypergraph Partitioning

Hypergraph partitioning is a generalization of traditional graph partitioning, where a hypergraph is a structure in which an edge (called a hyperedge) can connect more than two vertices (e.g., data items in our context). Hypergraph partitioning is a complex but powerful tool for optimizing relationships among entities represented in hypergraphs. In our context, the partitioner solves the following optimization problem:

$$\min_{\mathcal{P}} \quad \mathrm{cut}(H, \mathcal{P}) \quad \text{subject to} \quad \forall d_i \in \mathcal{P}, \ |V_{d_i}| \leq (1 + \epsilon) \frac{|V|}{k}$$

where $H(V, E)$ is the hypergraph representing the workload, $\mathcal{P} = \{d_1, d_2, \ldots, d_k\}$ is a partition of the vertex set V into k shards, $cut(H, \mathcal{P})$ denotes the total weight of hyperedges cut by the partition $\mathcal{P}$, and ϵ represents the imbalance tolerance factor.

The KaHyPar algorithm is chosen for its efficiency in handling large-scale hypergraphs, achieving near-linear time complexity in practical scenarios [72]. The time complexity of the hypergraph partitioning algorithm is $O(|E| \log k)$, where $|E|$ is the number of hyperedges, and k is the number of partitions.

The Partitioner also uses the Hungarian algorithm to minimize data movement for the shard reassignments. The Hungarian method is an optimization algorithm that efficiently solves assignment problems by finding the minimum cost matching in a bipartite graph [52]. Assuming that $\mathcal{P}' = \{d_1', d_2', \ldots, d_k'\}$ were the original shards, $\mathcal{P} = \{d_1, d_2, \ldots, d_k\}$ are the new shards, and $cost(d_i', d_j)$ denotes the cost of moving data from d_i' to d_j. It solves the assignment problem of mapping each shard in $\mathcal{P}'$ to exactly one shard in $\mathcal{P}$ while minimizing the total cost of data transfer.

4.3 Limitations

While the centralized architecture can tolerate malicious failures within clusters, it suffers from three main limitations.

First, the central coordinator lacks Byzantine fault tolerance. If the coordinator becomes faulty or compromised, it can disrupt the entire system by misrouting transactions or providing incorrect shard mappings. As mentioned earlier, achieving fault tolerance through the replication of the coordinator across a cluster of nodes is not particularly difficult. However, this would introduce additional complexity and overhead as the nodes in the coordinator committee must agree on the same global view of the system, which requires running a consensus protocol among them.

Second, the coordinator-based solution, whether utilizing a single node or a cluster of coordinator nodes, incurs considerable resource overhead. This is because resources must be allocated to the trusted administrative domain, which does not contribute to transaction processing and is solely dedicated to resharding purposes.

Finally, this design relies on a centralized component, which could be a potential target for attackers. The administrative domain oversees shard assignment and the overall processing of transactions; if it were to be compromised by a strong adversary, the system's performance could be significantly impacted.

5 Decentralized Architecture

To eliminate reliance on a central trusted component and enhance fault tolerance against Byzantine failures, we propose a fully decentralized architecture. In this architecture, as presented in Figure 5, shards autonomously perform data placement, performance monitoring, and resharding decisions. Nodes collaboratively coordinate these activities through decentralized interactions, effectively removing centralized coordination and improving resilience.

5.1 Architecture Overview

In the decentralized architecture, each node maintains a local data shard map consisting of entries of the form $\langle$ key ID $\rightarrow$ shard ID $\rangle$, associating each data key with its current residing shard. The data shard map size scales linearly with the number of keys in the system, but remains negligible relative to the actual data storage requirements. For example, in our experiments with 1 million keys, the shard map requires approximately 8 MB of storage per node, whereas each shard could possibly store gigabytes of actual key-value data. In general, the metadata represents less than 1% of the total storage overhead in typical deployment scenarios.
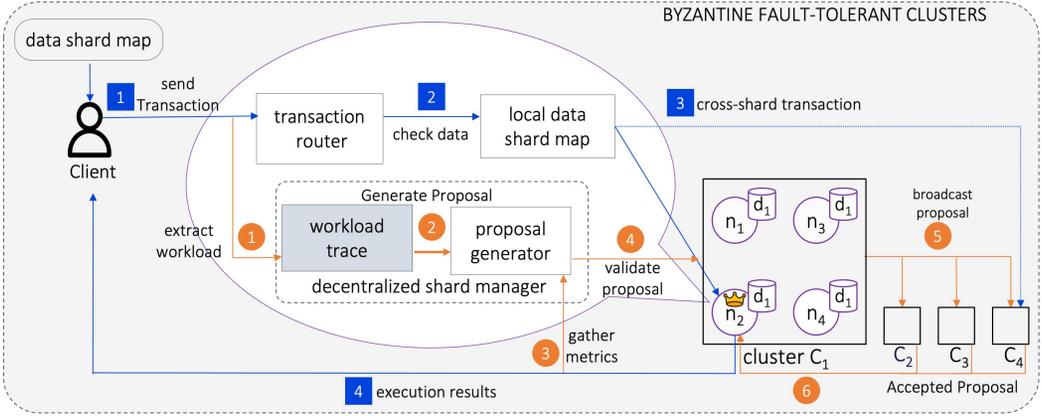
Fig. 5. Decentralized Architecture

To initiate a transaction, a client checks its local data shard map to identify the cluster containing the relevant data and sends the transaction to that cluster's leader ([1]). The transaction router on the leader parses the transaction and identifies its type using its local shard map ([2]). If the data resides within the same cluster, the transaction is executed locally using PBFT. Otherwise, the request is routed to the leaders of the involved clusters ([3]). Cross-shard transactions use the two-phase commit over PBFT, as presented in Figure 3. Once a transaction is executed, nodes send replies to the client ([4]) where the client updates its map if necessary.

5.2 Performance Monitoring

Each node's decentralized shard manager tracks local metrics including throughput, average latency, and the proportion of transactions involving multiple shards. When performance degradation is observed, such as latency exceeding a defined threshold or a persistent drop in throughput, the node considers initiating resharding.

To prevent unnecessary or overly frequent reshardings, nodes apply a moving average over recent transaction intervals. Resharding triggers only occur if the degraded performance persists across several intervals. Thresholds are also dynamically tuned based on workload characteristics, and the monitoring logic ignores extremely low-throughput phases to filter noise. Additionally, nodes operate under a constant rate injection model to stabilize throughput measurements and avoid attributing variation to client-side effects.

5.3 Key Affinity Resharding Algorithm

Upon detecting sustained performance issues within a cluster, the cluster leader proposes resharding using our *key affinity resharding* algorithm. This decentralized heuristic prioritizes keys frequently involved in cross-shard transactions, aiming to reduce communication overhead. Our mechanism assigns each candidate key a hybrid score that balances its overall cross-shard access frequency with its affinity to a particular shard. Importantly, the weighting of these factors adapts dynamically to the current workload; when the fraction of cross-shard transactions increases, the algorithm emphasizes co-locating keys that interact strongly with one shard (affinity), whereas when cross-shard activity is low, the priority shifts toward migrating hot spots to balance load.

Formally, for a key k on shard i, let f_k denote the number of recent cross-shard transactions involving k, and let $x_{k,j}$ denote the number of those transactions that access shard d_j (with $d_j \in \mathcal{P}$,

the set of all shards). We define the affinity score as:

$$a_{k,j} = x_{k,j} \cdot \left(0.2 + 0.8 \cdot \frac{x_{k,j}}{\sum_{m \in \mathcal{P}} x_{k,m}} \right)$$

Here, the term 0.2 provides a baseline contribution for each interaction, ensuring that every access counts, while the factor 0.8 amplifies the score when a key's interactions are concentrated with a single shard. This interpolation between pure frequency and exclusive interaction was derived empirically, similar to adaptive techniques observed in systems such as Amazon DynamoDB [5] and Google Spanner [31] and has been further validated in studies on elastic partitioning and adaptive resharding [4].

We then compute a final score for key k considering both its migration necessity and its localized affinity:

$$s_k = w_f \cdot f_k + w_a \cdot \max_{j \in \mathcal{P}} a_{k,j}$$

The weights w_a and w_f are set adaptively based on global cross-shard transaction percentage p:

$$w_a = 0.4 + 0.6p, \quad w_f = 1.0 - 0.5 w_a$$

The choice of 0.4 in w_a ensures that even when cross-shard transactions are minimal (i.e., $p \approx 0$), affinity contributes meaningfully to the key's score. The additional 0.6 scales the weight such that under high cross-shard traffic (i.e., $p \approx 1$), w_a approaches 1.0; in turn, the definition $w_f = 1.0 - 0.5 w_a$ guarantees that the frequency component is never completely ignored. This maintains a balance between migrating frequently accessed keys (hot spots) and preserving cohesive key-to-shard relationships. These parameter values were determined through extensive simulation and can be tuned for specific environments, as also suggested in prior adaptive load-balancing work [75].

To limit overhead and maintain stability during resharding, we enforce several constraints. First, a *per-destination cap* limits the number of keys migrating to a single shard per round. Second, a *per-round cap* restricts the total number of key migrations per shard. Finally, the quorum certificate requires $2f + 1$ signatures from both source and destination clusters before executing a migration, to prevent malicious nodes from making invalid resharding decisions.

These measures enable safe, local, and incremental proposal generation using only local logs and views. In effect, our two-tier approach, where the affinity score is computed for each key/shard pair and blended with overall transaction frequency using adaptive weights, ensures that resharding decisions are made by up-to-date workload conditions, with parameter values grounded in empirical observations and well-established adaptive re-balancing techniques.

5.4 Proposal Generation

Once a shard identifies sustained performance issues, the shard autonomously generates resharding proposals based on recent workload metrics and affinity scores, as illustrated in Figure 5 and Algorithm 1. In particular, workload extraction (❶) analyzes recent transaction logs for scoring keys and proposal generation (❷) ranks keys based on computed affinity scores. Each resharding proposal contains the following components: (1) *key sets*: one or more sets of keys proposed for migration, where keys within each set exhibit strong affinity through frequent co-access in transactions, (2) *source cluster*: the proposing cluster id, (3) *destination cluster*: the target cluster id, (4) *performance metrics*: justification data including the frequency of cross-shard transactions involving the proposed keys, current load imbalance indicators, and expected performance improvement statistics, and (5) *quorum certificate*: a collection of $2f + 1$ signatures from nodes in the proposing cluster, validating local consensus on the proposal. For example, a proposal from cluster C_1 might specify the migration of keys $\{k_1, k_3, k_7\}$ to cluster C_2, accompanied by statistics indicating that these keys generate 80% of C_1's cross-shard transaction traffic with C_2. Keys are grouped within

---

Algorithm 1 Decentralized Resharding with Key Affinity

Require: LocalMetrics, LocalLogs, ShardMap
Ensure: Atomic resharding if proposal passes consensus
 1: **procedure** MONITORPERFORMANCE()
 2:     **if** latency > threshold **or** throughput drops persistently
 3:         GENERATEPROPOSAL()
 4:     **end if**
 5: **end procedure**

 6: **procedure** GENERATEPROPOSAL()
 7:     Extract f_k and $x_{k,j}$ from LocalLogs
 8:     Compute $a_{k,j} = x_{k,j} \cdot (0.2 + 0.8 \cdot \frac{x_{k,j}}{\sum_m x_{k,m}})$
 9:     Compute $s_k = w_f \cdot f_k + w_a \cdot \max_j a_{k,j}$
10:     Select top keys where $j^* = \arg\max_j a_{k,j} \neq i$
11:     Simulate impact; discard if unhelpful
12:     Broadcast to local cluster for signatures
13:     **if** receiving a quorum certificate ($2f + 1$ signatures) **then**
14:         NEGOTIATEWITHDESTINATIONS()
15:     **end if**
16: **end procedure**
17: **procedure** NEGOTIATEWITHDESTINATIONS()
18:     **for** every destination shard:
19:         Simulate proposed migrations
20:         Accept subset if beneficial
21:         Sign and return accepted keys
22:     Aggregate into $\mathcal{M}_{\text{final}}$
23:     EXECUTEPLAN($\mathcal{M}_{\text{final}}$)
24: **end procedure**
25: **procedure** EXECUTEPLAN($\mathcal{M}_{\text{final}}$)
26:     **Phase 1: Prepare**
27:         Lock keys and verify state
28:         Abort if inconsistency is found
29:     **Phase 2: Commit**
30:         Transfer key-value data and update ShardMap
31:         Release locks and confirm commit
32: **end procedure**

---

proposals only when they demonstrate frequent co-occurrence in transactions, ensuring that related data items remain co-located after migration. The leader node then internally simulates the proposal effects on local metrics, discarding proposals that degrade performance or violate shard balance. If the simulation results are acceptable, the leader broadcasts a signed proposal message to all replicas within its cluster for validation before sending it to other clusters. Each proposal message includes multiple sets of highly cohesive data items, i.e., data items within each set have strong affinity. The leader assigns order to proposal messages, and proposal messages are processed using the utilized BFT consensus protocol of the cluster in the same way as transactions.

5.5 Proposal Validation

Proposals undergo an internal validation within the corresponding cluster (④), before they are shared with other clusters. Once a replica receives a proposal message from the leader replica *within* the same cluster, the replica independently simulates the proposed changes against its local transaction history. Each replica signs the proposal only if the updated scheme reduces cross-shard transactions and maintains shard balance. The cluster then uses the employed BFT consensus protocol to process the proposal. By relying on BFT protocols, MARLIN is able to prevent malicious

nodes from triggering resharding attacks. In particular, and as part of the consensus protocol, each proposal must obtain signatures (i.e., signed commit messages) from at least $2f + 1$ replicas (called *quorum certificate*) to be considered as valid. Given that at most f (out of $3f + 1$) nodes are assumed to be faulty, a quorum certificate ensures that the proposal is agreed upon by a majority of honest replicas, preventing malicious nodes from initiating safety attacks; even if a malicious leader generates a fraudulent proposal by reporting incorrect performance metrics and colludes with all other malicious nodes to accept the proposal, the system remains safe. As a result, dishonest behavior does not affect correctness or trigger unsafe resharding decisions. However, since nodes need to spend time validating such incorrect proposals, the system will face some performance degradation. As future work, we can add a feature where patterns of frequent unjustified re-sharding requests can be logged and monitored for early detection.

Moreover, when a replica detects a performance issue (using its local performance metrics), it starts its view-change timer, which will be stopped (or restarted if there are unexecuted transactions, following the logic of PBFT timers) when the replica receives a proposal message from the leader. If a leader fails to send a resharding proposal while the shard is experiencing ongoing performance issues, the timer on the replicas will expire, triggering the view-change routine. This mechanism helps in detecting a malicious leader that avoids sending valid resharding proposals to other nodes.

Once consensus is reached, the leader collects the quorum certificate (e.g., a quorum of $2f + 1$ signed commit messages) to use as proof of proposal validity in communication with other clusters.

5.6 Proposal Evaluation

Validated proposals along with a quorum certificate (i.e., $2f+1$ commit messages) are then broadcast to destination shards (⑤). The leader replica of the destination shard validates the proposal, assigns a sequence number to the proposal, and initiates a consensus instance. While multiple proposals might arrive at a cluster concurrently, proposals are sequentially processed using the internal BFT consensus of the cluster, ensuring deterministic ordering. Each node in a destination shard independently simulates the impact of proposed migrations on its local performance. If the changes are beneficial, i.e., reducing cross-shard transactions and preserving balance, the node votes for the proposal through the initiated consensus protocol (e.g., by sending signed prepare and commit messages in PBFT). If a quorum of $2f + 1$ signatures (e.g., signed commit messages) is collected, the shard returns the accepted subset to the source cluster (⑥).

The negotiation step allows each destination shard to selectively accept key migrations it finds beneficial, contributing to a resharding plan that reflects collaborative decisions made across the system. As a result, the final plan reflects not only the initiator's observations but also the preferences and constraints of all affected shards. Note that the destination shard is not able to accept random keys. As discussed in Section 5.4, each proposal message includes multiple sets of highly cohesive data items, and the destination shard can accept or reject a whole set. That way, keys with a strong affinity will be located in the same cluster.

5.7 Executing Resharding

The initiator aggregates all accepted data item sets (within the proposal message) into a final plan, which is executed across the source and destination clusters using a two-phase atomic commit protocol. The source cluster plays the coordinator role and initiates the prepare phase. Both source and destination shards lock relevant keys and validate local state. If any inconsistency is found, the cluster sends an abort vote. Upon receiving a quorum of $2f + 1$ prepared responses from the destination cluster, the source cluster broadcasts a commit vote after reaching consensus within the cluster. At this stage, keys are transferred, shard maps are updated, and finally, locks on records are released. This execution step finalizes the resharding, ensuring that keys are transferred atomically

and that all replicas converge on the new mapping. The system guarantees that a migration is atomic (i.e., either fully completes or has no effect), preserving safety even in the presence of faulty nodes.

Algorithm 1 summarizes the end-to-end decentralized resharding workflow, embedding the affinity-based scoring and validation steps within the broader execution flow.

6 Experimental Evaluation

We evaluate MARLIN across multiple configurations and workloads to answer the following questions:

- **RQ1. Performance.** How do the centralized and decentralized architectures perform under a range of workloads, and how do they compare to a system with no adaptive re-sharding?
- **RQ2. Adaptability.** How effectively does MARLIN adapt to workload changes over time, and how quickly does it stabilize?
- **RQ3. Overhead.** What is the runtime cost of re-sharding (e.g., throughput drop, latency impact, resource usage), and how do mechanisms like throttling or different resharding strategies affect this overhead?

6.1 Experimental Setup

We deploy MARLIN on a cluster of 16–128 physical nodes (depending on the experiment), each with 16-core Intel Xeon Platinum 8280 CPUs, 64–128 GB RAM, and 10 Gbps NICs. All nodes run Ubuntu 22.04 LTS and replicate data using Redis 6.2. Unless otherwise specified, shards are fully replicated across four nodes, enabling tolerance for $f = 1$ Byzantine faults per shard. Our default setup uses 4 shards (16 nodes), with scalability experiments extending to 32 shards (128 nodes). Our deployment supports the following configurations: (1) the centralized architecture, where a centralized (global) coordinator uses a hypergraph-based partitioner, (2) the decentralized architecture, where there is no centralized coordinator and nodes locally use a key affinity algorithm, and (3) a fixed, predefined (no resharding) architecture, which is our static baseline. Our baseline configuration ("no resharding") employs the same system architecture (PBFT within each cluster and 2PC for cross-shard transactions) without the adaptive resharding feature; data is partitioned once initially using static range partitioning and remains fixed throughout the experimental duration. This design enables direct comparison, isolating the impact of adaptivity, as both the baseline and our adaptive system incur identical consensus and transaction processing overhead. We selected this coordinator-based static baseline because our system employs a coordinator-based architecture, ensuring fair comparison conditions. Alternative baselines such as SharPer [8] utilize different transaction processing models (flattened) that would introduce additional variables into the experimental comparison.

We extend the Yahoo! Cloud Serving Benchmark (YCSB) [30] to support cross-shard, multi-key transactions, dynamic skew, and real-time resharding common in cloud-native applications and OLTP deployments. We evaluate across the following profiles:

- A: Balanced reads/writes (50/50), 10% cross-shard.
- B: Update-heavy (5% reads, 95% writes), 40% cross-shard.
- C: Read-only (100% reads), 25% cross-shard.
- D: High contention writes (50/50), 60% cross-shard.
- E: Scan-heavy, moderate cross-shard.

We have 1 million records where each record is 1 KB in size; transactions access 2–6 keys, with a 70% Zipfian skew to emulate hotkey contention (other than the dynamic workloads experiments (Section 6.3) where we measure the performance under different Zipfian skew parameters). All experiments are run for at least 15 minutes following a warm-up phase. Each point represents the
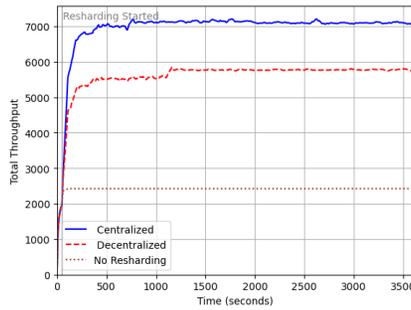
Fig. 6. Throughput of MARLIN under a static workload (YCSB-A) over time

average across 3 runs with 95% confidence intervals. Throughput includes committed and aborted transactions to reflect end-to-end performance.

6.2 RQ1. Performance Under Static Load

We first measure system throughput under a long-running static workload to compare the baseline performance of each architecture. Using YCSB Workload A (balanced read/write, 10% cross-shard), we run each configuration of MARLIN for one hour and track throughput over time. This extended run allows the adaptive schemes to converge to a stable partitioning and reveals steady-state performance after any re-sharding. Figure 6 presents the results.

The centralized architecture rapidly converges to a steady-state throughput of 7100 transactions per second, benefiting from its global view and immediate hypergraph optimization. This provides early performance gains in static workloads. The decentralized architecture, on the other hand, converges to the throughput of 5800 transactions per second, around 18% lower than the centralized architecture. This is because in the decentralized architecture, nodes do not have access to a global view, and resharding happens based on local observations.

The decentralized architecture, however, still demonstrates 2.5× throughput compared to the predefined fixed partitioning (no resharding). This result demonstrates the need for adaptivity in scalable data management systems, even if the percentage of cross-shard transactions is not high.

Transient performance dips occur due to the overhead of distributed resharding. The temporary drop in throughput seen in the decentralized architecture (around 500–1000 seconds) happens when parallel migrations begin. This dip is not a sign of algorithm instability; instead, it reflects the costs of constructing quorums, data movement, and coordination. Once the resharding is complete, performance improves and stabilizes.

These results suggest that while centralized partitioning provides a short-term boost in static settings, decentralized MARLIN is a viable alternative that catches up quickly, even with no global coordination. This makes it particularly attractive in large-scale untrustworthy deployments, where centralized control may be impractical or undesirable. The fact that local-only heuristics can approximate global decisions over time emphasizes the strength of our affinity-aware resharding design.

This result also underscores the importance of adaptive sharding in achieving high throughput, particularly in workloads with a substantial percentage of cross-shard transactions. While centralized resharding benefits from global visibility and gains an early advantage, decentralized MARLIN, driven only by local affinity heuristics, closes most of the gap without requiring a central coordinator.
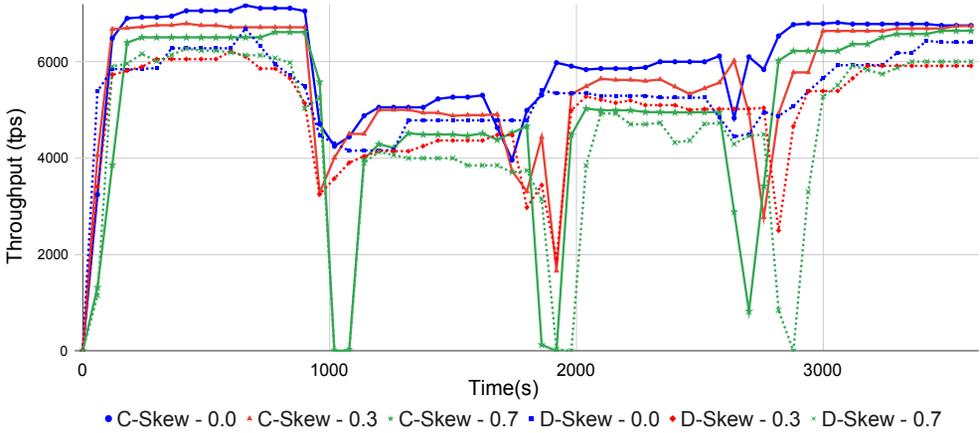
Fig. 7. Throughput during dynamic workload transitions. Vertical lines indicate phase changes.

6.3 RQ2. Adaptability to Dynamic Workloads

Next, we evaluate how effectively MARLIN adapts to changing workloads. This scenario simulates a dynamic deployment where workload patterns evolve over time, common in bursty cloud services, seasonal demand, or multi-tenant environments.

We rotate through three YCSB workloads every 900 seconds (15 minutes), for a total of one hour. The workloads are: A: Balanced-read write with 10% cross-shard transactions, B: Update-heavy with 40% cross-shard transactions, C: Read-only with 25% cross-shard transactions, and then returning to A for the final phase. This sequence creates a dynamic workload with varying contention and access patterns. Each workload (A, B, and C) employs a Zipfian key access distribution with an identical skew factor, but the identity of the hotkeys differs between workloads to further evaluate the system's adaptive resharding ability. We run the experiments for three different skew parameters (θ=0, θ=0.3 and θ=0.7) to evaluate the impact of workload skewness on the adaptivity of the system. Figure 7 tracks throughput as the workload shifts. Transitions are unannounced, and the system infers and adapts based solely on internal performance metrics.

Changing the workload skewness from 0.0 to 0.7 has a higher impact on the performance of the centralized architecture (3-10% reduction) compared to the decentralized architecture (2-6% reduction). This outcome is anticipated, as the overhead associated with processing workloads that exhibit higher skewness is partially covered by the overhead inherent to the decentralized architecture. The decentralized architecture (dashed lines) detects changes 40–60 seconds after a transition and begins gradually migrating keys. The centralized architecture (solid lines), on the other hand, reacts faster (25–30 seconds), but migrates in larger batches, leading to steeper (but shorter) throughput dips. In both cases, throughput recovers within 2–3 minutes of the shift. Minimum throughput during transitions dips to 3300 tps in the decentralized and 3000 tps in the centralized architecture. However, decentralized architecture retains a smoother curve due to staggered migrations, while the centralized architecture sees more abrupt drops and recoveries.

As can be seen, the throughput of both centralized and decentralized architectures in the final workload A is slightly different from the initial workload A due to three primary factors. First, as discussed earlier, while both the initial and final workload A maintain the same characteristics,
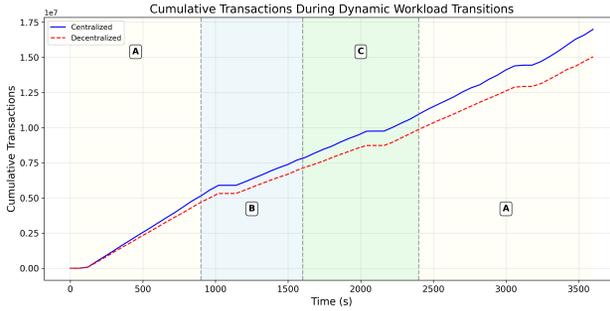
Fig. 8. Cumulative transactions during workload shifts.

the identity of the hotkeys differs (as they are selected randomly). Second, the centralized algorithm accumulates knowledge from the complete sequence of workloads; hence, the hypergraph partitioner produces a partitioning that balances the requirements of all observed workload types rather than optimizing exclusively for the current workload A. This represents a rational strategy to accommodate potential future workload variations. Finally, both systems avoid aggressive reversion to the initial partitioning because they anticipate continued workload variability. The centralized coordinator maintains a partitioning that minimizes worst-case performance across all observed patterns, while the decentralized approach cannot execute large reconfigurations rapidly due to its consensus requirements. This behavior prevents oscillatory behavior and provides stable performance characteristics.

Figure 8 demonstrates the cumulative transactions of the system when processing dynamic workloads presented in Figure 7 with a skew parameter equal to 0.7 As shown, the centralized architecture processes in total 16.64 M transactions, while the decentralized architecture handles 15.04 M (90.4% of the centralized architecture) by the end of the run; presenting a consistent performance for both architectures in the presence of workload shifts.

This experiment illustrates the adaptability of both architectures under dynamic load. The centralized architecture is faster to initiate change and ultimately attains slightly higher peak throughput, but the decentralized architecture recovers faster and exhibits less disruption. The results support our design choice to prioritize lightweight, local adaptation while retaining the ability to converge to efficient placements over time.

6.4 RQ3. Resharding Overhead

In this set of experiments, we isolate the overhead of resharding by measuring resource utilization and latency during resharding events. These metrics quantify the runtime cost of adaptation and show how system performance is affected during transitions.

Figure 9 shows that each resharding event introduces short-lived but bounded spikes across all resource dimensions. CPU usage increases by roughly 45%, caused by key migration and replica synchronization, and network usage peaks at 250–300 Mbps due to key movements. These spikes are short-lived and confined to resharding windows. Outside those windows, the system returns to baseline quickly and remains stable until the next shift. Similarly, transaction latency increases during resharding events, with spikes reaching up to 2.6× the baseline during migration windows before returning to normal levels. The decentralized architecture exhibits consistently lower mean latency than the centralized approach due to its incremental, less disruptive resharding behavior. Our 99th percentile measurements reveal that the decentralized architecture achieves 46 ms compared
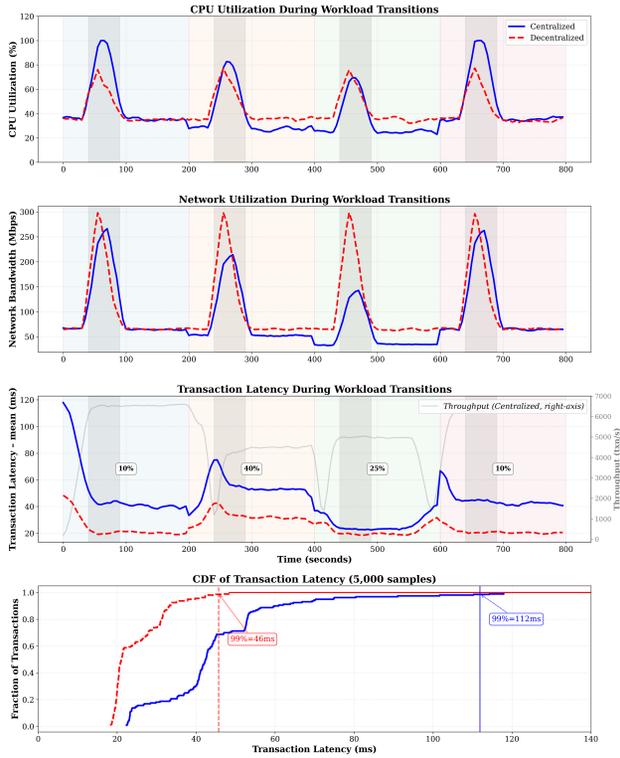
Fig. 9. CPU, network, and latency spikes during resharding. Shaded areas mark resharding windows.

to 112 ms for the centralized architecture, a 59% lower tail latency. The correlation between latency spikes and throughput reductions confirms that the decentralized system's lower latency results from less aggressive resharding behavior. During resharding windows, the centralized system experiences steeper throughput reductions (up to 15%) concurrent with the higher latency spikes, while the decentralized system maintains more stable performance characteristics. This demonstrates that the decentralized approach trades some optimization potential for operational stability.

In summary, resharding imposes brief, isolated overhead without long-term degradation. Unlike background approaches that spread cost across time, MARLIN front-loads migration work and stabilizes rapidly. This design minimizes overall disruption, avoids thrashing, and ensures predictable recovery behavior.

6.5 Ablation Studies

In this section, we conduct ablation studies to assess the impact of different components on the system's performance. We focus on two main factors: the percentage of cross-shard transactions in the workload and the number of clusters in the system. We run the two ablation studies on YCSB Workload B(update-heavy with 40% cross-shard transactions) with 4 shards and 16 nodes unless noted otherwise.

6.5.1 Impact of Cross-Shard Transaction Rate. To assess the effectiveness of the resharding mechanisms used in MARLIN, we examine the impact of varying the percentage of cross-shard transactions
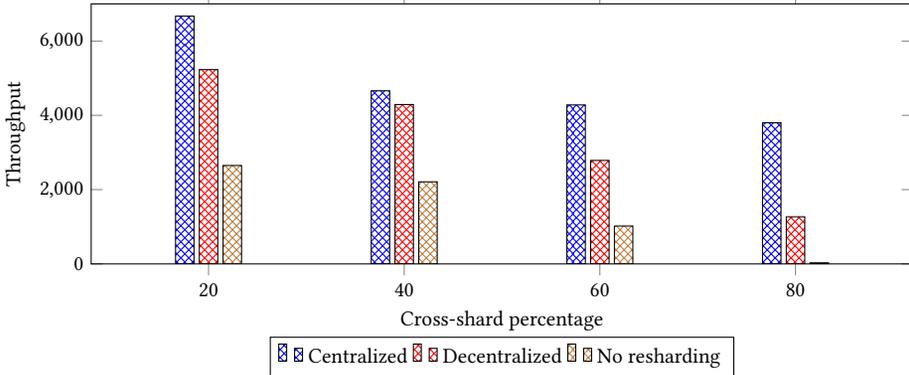
Fig. 10. Throughput vs Cross-Shard Percentage.

on system throughput. We vary the cross-shard transaction rate in the workload from 20% to 40%, 60%, and 80%.

We maintain the default system configuration and measure the average throughput for each cross-shard percentage using both the centralized architecture and the decentralized architecture. The no-resharding configuration serves as a baseline for comparison.

Figure 10 shows a clear downward trend in throughput of all three different configurations as cross-shard percentage increases, driven by coordination overheads.

As can be seen, the centralized architecture achieves the highest throughput across all three configurations. In a workload with 20% cross-shard transactions, the centralized architecture is able to process 6.7 ktps. Increasing the percentage of cross-shard transactions to 40% reduces the throughput of the centralized architecture to 4.7 ktps, demonstrating a 30% decrease in throughput. A further increase in the percentage of cross-shard transactions results in a graceful degradation of the throughput.

The decentralized architecture, while demonstrating a competitive throughput in workloads with 20% and 40% cross-shard transactions (21% and 9% lower throughput, respectively, compared to the centralized architecture), faces a significant reduction in workload with a higher percentage of cross-shard transactions (60% and 80%). This is expected because in such workloads, different clusters continuously generate and broadcast data resharding proposals, preventing the system from processing transactions. Note that the decentralized architecture still outperforms the no-resharding baseline by 1.5–2.6× as the cross-shard rate increases from 20% to 60%. The no-resharding baseline suffers from poor performance beyond 60% cross-shard transactions, with throughput dropping to zero.

These results reinforce that adaptive resharding, particularly using workload-aware partitioning, is critical for sustaining throughput under high contention and cross-shard activity. The ability to co-locate frequently accessed keys provides measurable benefits, especially in skewed or multi-tenant environments.

6.5.2 Impact of Number of Clusters. We assess the scalability of Marlin by evaluating the system's throughput as we vary the number of clusters. To achieve this, we vary the number of clusters from 2 to 4, 8, 16, and 32 while keeping other parameters at their default values. We measure the average throughput of the centralized and decentralized architectures and plot the results against the number of clusters. We also include a configuration without adaptive resharding to serve as a baseline.
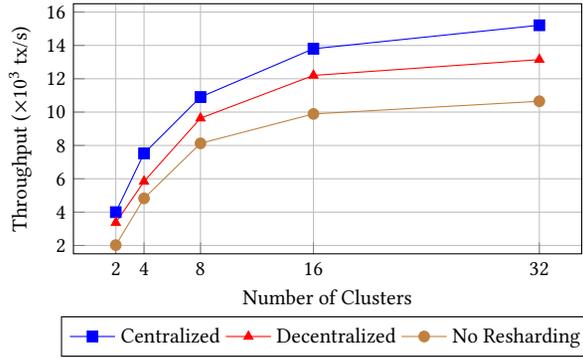
Fig. 11. Throughput vs. number of clusters.

As shown in Figure 11, both centralized and decentralized architectures scale well with the number of clusters; however, as the number of clusters increases, communication overhead and resource limitations lead to diminishing returns in performance improvements. The centralized architecture consistently achieves (13% to 29%) higher throughput due to its global optimization of data placement, which reduces cross-shard transactions and balances the workload more efficiently. The decentralized architecture also scales well but attains slightly lower throughput, attributed to its reliance on local information, which may lead to suboptimal shard placements. The no-resharding configuration exhibits the lowest throughput (e.g., 16% to 40% lower than the decentralized architecture), highlighting the importance of adaptive resharding in distributed systems.

6.6 Evaluation Summary

Across various configurations, we observe that Marlin maintains high throughput under both static and dynamic workloads, and remains stable during repeated re-sharding. Centralized architecture reaches optimal placement faster due to its global view, but it begins to incur coordination overhead at scale. Decentralized architecture, while relying only on local metrics, achieves comparable steady-state performance and reacts efficiently to workload shifts with lower disruption. Under adversarial conditions, the system maintains correctness and avoids unsafe re-sharding using quorum certificates (i.e., a quorum of $2f + 1$ signatures). The takeaways of our experimental evaluation can be summarized as follows.

- **Adaptivity yields the largest gains:** Given the same data partitioning, centralized and decentralized architectures demonstrate almost similar throughput; removing the coordinator does not fundamentally reduce performance. The primary performance gains come from being able to adapt the partitioning (versus a static design), not from centralized control.
- **Near-optimal partitioning in practice:** A globally optimized hypergraph cut provided 50% higher throughput than a non-resharding configuration in our tests, and our decentralized heuristic nearly matched that benefit. This confirms that our lightweight heuristic successfully captures the important workload structure and approaches the effectiveness of heavy-duty graph partitioning.
- **Better scalability without a coordinator:** The decentralized architecture avoids the coordinator bottleneck and continues to scale almost linearly at 64 nodes, whereas the centralized architecture begins to hit limits. This shows that removing the single coordinator can be

advantageous in very large systems or in environments where no single trusted coordinator is available.
- **Controlled overhead:** The overhead of re-sharding is manageable. By tuning the frequency and scope of resharding, Marlin keeps throughput and latency stable over time. Our on-demand, rate-limited re-sharding approach allows the system to adapt to major workload changes without excessive thrashing.

In summary, Marlin bridges the gap between prior systems that employ dynamic sharding in trusted environments and static sharding in untrusted environments, providing a practical solution for an adaptive, high-performance data management in untrusted environments.

7 Related Work

Today's large-scale distributed data management systems need to deal with untrustworthy environments where multiple mutually distrustful entities communicate with each other, and maintain data on untrusted infrastructure. By relying on Byzantine fault-tolerant (BFT) protocols, distributed data management systems deployed in untrustworthy environments have enabled a large class of distributed applications ranging from contact tracing [66], crowdworking [9], supply chain assurance [11, 83], and federated learning [67]. Byzantine fault tolerance refers to servers that behave arbitrarily after the seminal work by Lamport, et al. [56].

Partitioning the data into multiple shards that are maintained by different subsets of nodes is a proven approach to enhance the scalability of databases. Data sharding techniques are commonly used in globally distributed databases such as H-store [48], Calvin [81], Spanner [31], Scatter [42], Google's Megastore [19], Amazon's Dynamo [34], Facebook's Tao [26], and E-store [80]. In such systems, nodes are assumed to be trusted, and a coordinator node is used to process crash-shard transactions.

Sharding techniques have also been used in distributed data management systems deployed in untrusted environments, e.g., permissioned blockchains. Scalability can be achieved using transaction sharding, e.g., ResilientDB [45], or data sharding techniques, e.g., AHL [33], SharPer [8], Saguaro [10], and Multi-channel Fabric [14]. In the transaction sharding technique, the entire data is replicated on all clusters, while each cluster is responsible for running consensus on a separate subset of transactions. Using data sharding techniques, on the other hand, the data is partitioned into multiple shards that are maintained by different clusters. Such systems process two types of intra-shard and cross-shard transactions, where cross-shard transactions can be processed either in a centralized manner using the coordinator-based approach or in a decentralized manner using the flattened approach.

Data sharding approaches mainly differ in how they process cross-shard transactions. Centralized processing of cross-shard transactions is simpler and closer to the traditional two-phase commit, i.e., instead of a single coordinator node, as shown in Section 3, a coordinator cluster is needed to tolerate Byzantine failure. However, coordinator-based approaches require a large number of intra- and cross-cluster communication phases. On the other hand, the decentralized approach does not require an extra set of nodes, processes transactions in fewer phases among the involved clusters, and is able to process cross-shard transactions with non-overlapping clusters in parallel. However, if the involved clusters are distant, establishing cross-shard consensus among involved clusters requires multiple rounds of message passing, resulting in high latency. Transaction sharding approaches, on the other hand, do not suffer from the latency of processing cross-shard transactions by replicating the entire data on every cluster. However, exchanging messages between all clusters for every single transaction still results in high latency.

All such scalable systems, while tolerating Byzantine failures, typically rely on a fixed, predefined sharding scheme and do not reshard data adaptively, in response to workload changes.

Meeting the diverse requirements of distributed applications necessitates data management systems to be not only secure but also adaptive to accommodate shifting workloads, trust assumptions, and deployment environments [84]. Consequently, traditional distributed data management systems encounter challenges at every layer of the system stack. Although there have been efforts to design systems that enhance adaptability within specific components, such as BFT consensus protocols [16, 44, 86, 87] or transaction management paradigms [85], the issue of adaptive sharding in untrusted environments remains unaddressed.

Several systems have proposed workload-driven adaptive sharding strategies to improve performance and reduce coordination. For example, Schism [32] uses graph partitioning to minimize distributed transactions, while SWORD [69] explores tenant-based logical partitioning. However, these techniques assume a trusted deployment setting and do not account for adversarial behavior. Moreover, they are optimized for relatively static workloads and require centralized coordination for partitioning decisions.

In contrast, MARLIN supports dynamic resharding under Byzantine faults, and operates without relying on a centralized controller (in its decentralized architecture). In summary, compared to data management systems deployed in untrusted environments, MARLIN dynamically distributes data across different shards to improve performance in response to workload changes. Compared to adaptive data management systems deployed in trusted environments, MARLIN does not rely on a centralized coordinator, tolerates Byzantine failures, and performs resharding in a decentralized manner.

8 Conclusion

MARLIN is a scalable data management system specifically designed for untrustworthy environments, enabling efficient adaptation to dynamic workloads. The system utilizes Byzantine Fault Tolerance (BFT) consensus protocols to achieve agreement within each cluster, while employing a coordinator-based BFT atomic commitment protocol (based on two-phase commit) to process cross-shard transactions. We present centralized and decentralized architectures. The centralized architecture, which functions as a baseline, employs hypergraph partitioning and relies on a trusted administrative domain to facilitate efficient shard management. In contrast, the decentralized architecture eliminates the need for a central trusted component, where using a key affinity resharding algorithm, nodes collaboratively establish shard assignments in a decentralized fashion.

Our evaluation shows that MARLIN maintains consistent throughput and scalability across a range of workload scenarios. The centralized architecture adapts quickly by leveraging global workload knowledge, while the decentralized architecture provides resilience and scalability, albeit with slower convergence due to its partitioning strategy. This highlights a fundamental trade-off between performance optimality and system decentralization.

In future work, we plan to extend MARLIN to support heterogeneous workloads, explore more sophisticated partitioning strategies for the decentralized setting, and incorporate adaptive mechanisms [84–87] that better capture temporal workload patterns. We are also interested in evaluating MARLIN across a broader range of deployment settings, including variable-bandwidth environments and OLTP-style benchmarks. Finally, we aim to formalize the security guarantees of our design and expand adversarial testing to include a wider class of Byzantine behaviors.

Acknowledgments

References

[1] [n. d.]. Central Bank Digital Currency (CBDC). https://www.hkma.gov.hk/eng/key-functions/international-financial-centre/fintech/central-bank-digital-currency/

[2] [n. d.]. The Open Access AI Cloud. https://hyperbolic.xyz

[3] [n. d.]. SuiPlay. https://suiplay.sui.io

[4] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J Elmore. 2011. Database scalability, elasticity, and autonomy in the cloud. In *Int. conf. on Database Systems for Advanced Applications (DASFAA)*. Springer, 2–15.

[5] Amazon Web Services. 2018. How Amazon DynamoDB Adaptive Capacity Accommodates Uneven Data Access Patterns (or Why What You Know About DynamoDB Might Be Outdated). https://aws.amazon.com/blogs/database/how-amazon-dynamodb-adaptive-capacity-accommodates-uneven-data-access-patterns-or-why-what-you-know-about-dynamodb-might-be-outdated/. Accessed: 2025-04-16.

[6] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: a cross-application permissioned blockchain. *Proc. of the VLDB Endowment* 12, 11 (2019), 1385–1398.

[7] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. On Sharding Permissioned Blockchains. In *Int. Conf. on Blockchain*. IEEE, 282–285.

[8] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *SIGMOD Int. Conf. on Management of Data*. ACM, 76–88.

[9] Mohammad Javad Amiri, Joris Duguépéroux, Tristan Allard, Divyakant Agrawal, and Amr El Abbadi. 2021. SEPAR: Towards Regulating Future of Work Multi-Platform Crowdworking Environments with Privacy Guarantees. In *The Web Conf. (WWW)*. 1891–1903.

[10] Mohammad Javad Amiri, Ziliang Lai, Liana Patel, Boon Thau Loo, Eric Lo, and Wenchao Zhou. 2023. Saguaro: An Edge Computing-Enabled Hierarchical Permissioned Blockchain. In *Int. Conf. on Data Engineering (ICDE)*. IEEE, 259–272.

[11] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. 2022. Qanaat: A Scalable Multi-Enterprise Permissioned Blockchain System with Confidentiality Guarantees. *Proc. of the VLDB Endowment* 15, 11 (2022), 2839–2852.

[12] Mohammad Javad Amiri, Sujaya Maiyya, Daniel Shu, Divyakant Agrawal, and Amr El Abbadi. 2023. Ziziphus: Scalable Data Management Across Byzantine Edge Servers. In *Int. Conf. on Data Engineering (ICDE)*. IEEE, 490–502.

[13] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, and Yacov Manevich. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*. ACM, 30:1–30:15.

[14] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. 2018. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 111–131.

[15] Junaid Arshad, Muhammad Ajmal Azad, Alousseynou Prince, Jahid Ali, and Thanasis G Papaioannou. 2022. Reputable–a decentralized reputation system for blockchain-based ecosystems. *IEEE Access* 10 (2022), 79948–79961.

[16] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The next 700 BFT protocols. *Transactions on Computer Systems (TOCS)* 32, 4 (2015), 12.

[17] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. Medrec: Using blockchain for medical data access and permission management. In *Int. Conf. on Open and Big Data (OBD)*. IEEE, 25–30.

[18] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. 2019. Deploying intrusion-tolerant SCADA for the power grid. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 328–335.

[19] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. on Innovative Data Systems Research (CIDR)*.

[20] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep* (2019).

[21] Tara Siegel Bernard and Ron Lieber. [n. d.]. Banks Are Closing Customer Accounts, With Little Explanation. https://www.nytimes.com/2023/04/08/your-money/bank-account-suspicious-activity.html

[22] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: dependable and secure storage in a cloud-of-clouds. *Transactions on Storage (TOS)* 9, 4 (2013), 12.

[23] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2008. The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy* 6, 6 (2008), 44–51.

[24] Kenneth P Birman, Thomas A Joseph, Thomas Raeuchle, and Amr El Abbadi. 1985. Implementing fault-tolerant distributed objects. *Trans. on Software Engineering* 6 (1985), 502–508.

[25] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.

[26] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, and Harry Li. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Annual Technical Conf. (ATC)*. USENIX Association, 49–60.

[27] Yehonatan Buchnik and Roy Friedman. 2020. FireLedger: a high throughput blockchain consensus protocol. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1525–1539.

[28] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 173–186.

[29] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults.. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 9. USENIX Association, 153–168.

[30] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *symposium on Cloud computing (SoCC)*. 143–154.

[31] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, and Peter Hochschild. 2013. Spanner: Google's globally distributed database. *Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[32] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment* 3, 1 (2010).

[33] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *SIGMOD Int. Conf. on Management of Data*. ACM, 123–140.

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *Operating Systems Review (OSR)* 41, 6 (2007), 205–220.

[35] David DeWitt and Jim Gray. 1992. Parallel database systems: The future of high performance database systems. *Commun. ACM* 35, 6 (1992), 85–98.

[36] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. 2013. PoWerStore: Proofs of writing for efficient and robust storage. In *Conf. on Computer and communications security (CCS)*. ACM, 285–298.

[37] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[38] Ethan Frey and Christopher Goes. 2018. Cosmos Inter-Blockchain Communication (IBC) Protocol. https://cosmos.network.

[39] Miguel Garcia, Nuno Neves, and Alysson Bessani. 2013. An intrusion-tolerant firewall design for protecting SIEM systems. In *Conf. on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 1–7.

[40] Miguel Garcia, Nuno Neves, and Alysson Bessani. 2016. SieveQ: A layered bft protection system for critical services. *IEEE Transactions on Dependable and Secure Computing* 15, 3 (2016), 511–525.

[41] Danezis George and Sarah Meiklejohn. 2016. Centrally Banked Cryptocurrencies. In *Network and Distributed System Security Symposium (NDSS)*.

[42] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. 2011. Scalable consistency in Scatter. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 15–28.

[43] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. 2004. Efficient Byzantine-tolerant erasure-coded storage. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 135–144.

[44] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *European conf. on Computer systems (EuroSys)*. ACM, 363–376.

[45] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proceedings of the VLDB Endowment* 13, 6 (2020), 868–883.

[46] James Hendricks, Gregory R Ganger, and Michael K Reiter. 2007. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 73–86.

[47] Evan PC Jones, Daniel J Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD Int. Conf. on Management of Data*. ACM, 603–614.

[48] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, and Yang Zhang. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[49] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. 2013. Survivable SCADA via intrusion-tolerant replication. *IEEE Transactions on Smart Grid* 5, 1 (2013), 60–70.

[50] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Symposium on Security and Privacy (SP)*. IEEE, 583–598.

[51] K. Korpela, J. Hallikas, and T. Dahlberg. 2017. Digital supply chain transformation toward blockchain integration. In *Hawaii Int. Conf. on system sciences (HICSS)*.

[52] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.

[53] Jae Kwon. 2014. Tendermint: Consensus without mining. (2014).

[54] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

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

[56] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.

[57] Butler W Lampson and Howard E Sturgis. 1979. *Crash recovery in a distributed data storage system.* Xerox Palo Alto Research Center Palo Alto, California.

[58] Yimeng Liu, Yizhi Wang, and Yi Jin. 2012. Research on the improvement of MongoDB Auto-Sharding in cloud environment. In *Int. Conf. on Computer Science and Education (ICCSE)*. IEEE, 851–854.

[59] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM, 17–30.

[60] Dahlia Malkhi and Michael K Reiter. 1998. Secure and scalable replication in Phalanx. In *Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 51–58.

[61] Bhavana Mehta, Neelesh C. A, Prashanth S. Iyer, Mohammad Javad Amiri, Boon Thau Loo, and Ryan Marcus. 2023. Towards Adaptive Fault-Tolerant Sharded Databases (Extended Abstracts). In *Workshop on Applied AI for Database Systems and Applications (AIDB)*. CEUR-WS.org.

[62] Louise E Moser, Peter M Melliar-Smith, Priya Narasimhan, Lauren A Tewksbury, and Vana Kalogeraki. 1999. The Eternal system: An architecture for enterprise applications. In *Int. Enterprise Distributed Object Computing Conf. (EDOC)*. IEEE, 214–222.

[63] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[64] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A global-scale byzantizing middleware. In *Int. Conf. on Data Engineering (ICDE)*. IEEE, 124–135.

[65] André Nogueira, Miguel Garcia, Alysson Bessani, and Nuno Neves. 2018. On the challenges of building a BFT SCADA. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 163–170.

[66] Zhe Peng, Cheng Xu, Haixin Wang, Jinbin Huang, Jianliang Xu, and Xiaowen Chu. 2021. P2B-Trace: Privacy-Preserving Blockchain-based Contact Tracing to Combat Pandemics. In *SIGMOD Int. Conf. on Management of Data*. ACM, 2389–2393.

[67] Zhe Peng, Jianliang Xu, Xiaowen Chu, Shang Gao, Yuan Yao, Rong Gu, and Yuzhe Tang. 2021. Vfchain: Enabling verifiable and auditable federated learning via blockchain systems. *IEEE Transactions on Network Science and Engineering* (2021).

[68] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, et al. 2021. Bidl: A High-throughput, Low-latency Permissioned Blockchain Framework for Datacenter Networks. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 18–34.

[69] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. 2013. SWORD: scalable workload-aware data placement for transactional workloads. In *Int. Conf. on extending database technology (EDBT)*. 430–441.

[70] Noel Randewich and Arasu Kannagi Basil. [n. d.]. NYSE glitch sparks volatility in dozens of stocks. https://www.reuters.com/markets/us/nyse-equities-investigating-reported-technical-issue-2024-06-03/

[71] Tom Roeder and Fred B Schneider. 2010. Proactive obfuscation. *ACM Transactions on Computer Systems (TOCS)* 28, 2 (2010), 1–54.

[72] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics* 27 (2023), 1–39.

[73] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[74] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proc. of the VLDB Endowment* 10, 4 (2016), 445–456.

[75] Dalia Abdulkareem Shafiq, Noor Zaman Jhanjhi, Azween Abdullah, and Mohammed A Alzain. 2021. A load balancing algorithm for the data centres to optimize cloud computing applications. *Ieee Access* 9 (2021), 41731–41744.

[76] Xinmei Shen. [n. d.]. Tokenisation a defining trend for Web3 in Hong Kong as city hosts major Consensus event. https://www.scmp.com/tech/blockchain/article/3299190/tokenisation-defining-trend-web3-hong-kong-city-

hosts-major-consensus-event

[77] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2008. BFT Protocols Under Fire.. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 8. USENIX Association, 189–204.

[78] João Sousa and Alysson Bessani. 2015. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines. In *Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 146–155.

[79] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2009. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems* 21, 4 (2009), 452–465.

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

[81] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Int. Conf. on Management of Data*. ACM, 1–12.

[82] Catherine Thorbecke. [n. d.]. Silicon Valley Bank collapse sends tech startups scrambling. https://www.cnn.com/2023/03/10/tech/silicon-valley-bank-tech-panic/index.html

[83] Feng Tian. 2017. A supply chain traceability system for food safety based on HACCP, blockchain & Internet of things. In *Int. Conf. on service systems and service management (ICSSSM)*. IEEE, 1–6.

[84] Chenyuan Wu, Mohammad Javad Amiri, Haoyun Qin, Bhavana Mehta, Ryan Marcus, and Boon Thau Loo. 2024. Towards Full Stack Adaptivity in Permissioned Blockchains. *Proc. of the VLDB Endowment* 17, 5 (2024), 1073–1080.

[85] Chenyuan Wu, Bhavana Mehta, Mohammad Javad Amiri, Ryan Marcus, and Boon Thau Loo. 2023. AdaChain: A Learned Adaptive Blockchain. *Proc. of the VLDB Endowment* 16, 8 (2023), 2033–2046.

[86] Chenyuan Wu, Haoyun Qin, Mohammad Javad Amiri, Boon Thau Loo, Dahlia Malkhi, and Ryan Marcus. 2024. Towards Truly Adaptive Byzantine Fault-Tolerant Consensus. *ACM SIGOPS Operating Systems Review* 58, 1 (2024), 15–22.

[87] Chenyuan Wu, Haoyun Qin, Mohammad Javad Amiri, Boon Thau Loo, Dahlia Malkhi, and Ryan Marcus. 2025. {BFTBrain}: Adaptive {BFT} Consensus with Reinforcement Learning. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 1563–1583.

[88] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating agreement from execution for byzantine fault tolerant services. *Operating Systems Review (OSR)* 37, 5 (2003), 253–267.

[89] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 347–356.

[90] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling blockchain via full sharding. In *SIGSAC Conf. on Computer and Communications Security*. ACM, 931–948.

[91] Lidong Zhou, Fred Schneider, Robbert VanRenesse, and Zygmunt Haas. 2002. Secure distributed on-line certification authority. US Patent App. 10/001,588.

[92] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. 2002. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 329–368.