ADAPTIVE SHARDING AND FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

Bhavana Mehta

A DISSERTATION PROPOSAL

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

Ryan Marcus, Jonathan M Smith, Vincent Liu, Mohammad Javad Amiri

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2025

Supervisor of Dissertation

Boon Thau Loo, Professor of Computer and Information Science

Graduate Group Chairperson

Ryan Marcus, Assistant Professor of Computer and Information Science

Dissertation Committee

Jonathan M Smith, Professor of Computer and Information Science and Electrical Engineering
Vincent Liu, Associate Professor of Computer and Information Science
Mohammad Javad Amiri, Assistant Professor of Computer Science, Stony Brook University

ABSTRACT

ADAPTIVE SHARDING AND FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

Bhavana Mehta

Boon Thau Loo

Achieving scalability and reliability in distributed database systems often relies on effective data fragmentation and replication. However, traditional partitioning schemes assume trusted environments where nodes fail gracefully but do not act maliciously. In Byzantine environments, where nodes may behave adversarially, these assumptions no longer hold, presenting unique challenges to system design. This dissertation proposes Marlin, an adaptive fault-tolerant sharding framework designed to address these challenges.

Marlin introduces a dynamic sharding mechanism that optimizes data distribution for performance and fault tolerance in Byzantine environments. Two architectural paradigms are explored: a centralized architecture suited for trusted domains and a decentralized architecture that distributes shard management to eliminate single points of failure. Both designs employ real-time monitoring and adaptive algorithms to adjust sharding dynamically in response to workload patterns and adversarial conditions.

Preliminary evaluations demonstrate Marlin's effectiveness in maintaining high performance and resilience under diverse operational scenarios. By redistributing data adaptively, Marlin mitigates the impact of adversarial attacks, ensuring robust fault tolerance and adaptability to dynamic workloads. This proposal aims to advance the field of distributed systems by offering a scalable and reliable solution tailored for untrusted and adversarial environments.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Distributed systems rely on fault-tolerant protocols to provide robustness and high availability Birman et al. (1985); Moser et al. (1999); Corbett et al. (2013); DeCandia et al. (2007); Bronson et al. (2013); Kallman et al. (2008); Baker et al. (2011). While trusted cloud systems (e.g., Google's Spanner Corbett et al. (2013), Amazon's Dynamo DeCandia et al. (2007), Facebook's Tao Bronson et al. (2013)) rely on crash fault-tolerant (CFT) protocols, such as Paxos Lamport (2001), to establish consensus, trust-free systems (e.g., blockchains Amiri et al. (2022); Baudet et al. (2019); Kwon (2014); Amiri et al. (2023); Androulaki et al. (2018); Qi et al. (2021); Buchnik and Friedman (2020), lock servers Clement et al. (2009), certificate authority systems Zhou et al. (2002)) use Byzantine fault-tolerant (BFT) protocols to cope with untrustworthy environments.

Both CFT and BFT consensus protocols incur higher latency when more nodes are involved. Thus, trusted (e.g., Corbett et al. (2013); Taft et al. (2014b); Thomson et al. (2012)) and trust-free (e.g., Nawab and Sadoghi (2019); Amiri et al. (2021); Dang et al. (2019)) systems partition their data into *shards*, and replicate each shard on multiple nodes to provide scalability while guaranteeing fault tolerance. However, Byzantine environments introduce additional complexities and challenges, necessitating more robust solutions.

Approaches like SWORD Quamar et al. (2013) and Schism Curino et al. (2010) find optimal data fragmentations via hypergraph partitioning, with edges as transactions and tuples as vertices. Cutting the hypergraph into n pieces that break as few edges as possible while keeping each piece small enough to fit on a single node represents an optimal solution.

Unfortunately, what was optimal yesterday may not be optimal today: as workloads drift and new data is added, previously optimal fragmentation schemes can become arbitrarily poor. Several studies Marcus et al. (2018); Taft et al. (2014a) thus solve the problem in an adaptive way, moving data between fragments and moving fragments between nodes in an online fashion. Many of these are designed for analytical OLAP Hilprecht et al. (2020); Parchas et al. (2020) workloads and might not effectively adapt to transactional OLTP applications that frequently experience unpredictable demand shifts Taft et al. (2014a). To the best of our knowledge, none of these adaptive approaches consider Byzantine environments Zhou et al. (2023); Rzadca et al. (2020); Golovin et al. (2023). Byzantine environments bring a number of specific complications to the data fragmentation problem.

BFT vs. CFT Scalability. While all fragmentation schemes designed for transactional DBMSes aim to minimize the number of cross-shard transactions, the cost of a cross-shard transaction is significantly higher in an adversarial context. A round of Paxos Lamport (2001) consensus among n nodes requires $O(n)$ messages, but a round of PBFT Castro and Liskov (1999b) requires $O(n^2)$ messages. Special care is needed to avoid the quadratic cost of excessive cross-node transactions.

Special Constraints on Replication. A traditional distributed transactional database may choose to replicate data fragments to prevent data loss: to survive f nodes failing, data must be replicated $2f+1$ times. In a Byzantine environment, data must be replicated a minimum of $3f+1$ times to tolerate adversarial nodes. Additionally, a non-Byzantine system may label certain replicas as "primary" or "secondary," allowing transactions to only write changes to the "primary" replica. Such a strategy is impossible in a Byzantine environment: all nodes must participate in every transaction to ensure no adversarial node lies to the client.

Non-trustworthy Data. Traditional fragmented databases can accurately observe nodes required for each transaction. In a Byzantine environment, adversarial nodes could lie about a cross-node transaction to induce a particular fragmentation scheme. For example, an adversarial node could report that a set of tuples are never accessed together, causing a naive fragmentation scheme to separate those tuples into multiple nodes, potentially causing a denial-of-service attack.

One of the main challenges in sharded distributed protocols is finding an effective data partitioning approach that balances contradictory goals like load balancing, minimizing cross-shard transactions, and optimizing throughput and latency. Since the transaction workload is dynamic, such an approach must be adaptive, and repartitioning must be performed as needed. Existing adaptive sharding approaches for distributed databases focus primarily on crash failures Curino et al. (2010); Quamar et al. (2013) or fault-free environments. They aim to dynamically repartition data and assign shards to nodes based on workload characteristics and performance metrics but cannot handle Byzantine environments where nodes may exhibit arbitrary, potentially malicious behavior.

This proposal introduces Marlin, a scalable distributed database system designed to tolerate Byzantine faults while efficiently adapting to dynamic workloads. Marlin continuously refines shard boundaries based on real-time workload characteristics, minimizing cross-shard communication and maximizing system performance. By employing Practical Byzantine Fault Tolerance (PBFT) for both intra-shard and cross-shard transactions, Marlin achieves high availability and fault tolerance without relying on centralized coordination. It tolerates up to 33% Byzantine node failures, maintaining high performance and reducing communication overhead. Our contributions are:

Adaptive sharding for byzantine faults: We design and implement Marlin, introducing a novel adaptive sharding mechanism that dynamically adjusts data partitions in response to workload changes in Byzantine environments. This mechanism is resilient to malicious behaviors, ensuring efficient data distribution and load balancing.

Dual architecture exploration: We investigate both centralized and decentralized architectures for shard assignment within Marlin. The centralized architecture uses a trusted administrative domain for efficient shard management, while the decentralized architecture enables nodes to collaboratively determine shard assignments without trusted components, enhancing fault tolerance.

Robust byzantine fault handling: Marlin integrates PBFT for both intra-shard and cross-shard consensus, allowing the system to tolerate up to 33% Byzantine nodes. This ensures consistent transaction execution and data integrity, even in the presence of malicious actors.

Comprehensive evaluation: We conduct extensive experiments under various workloads and fault conditions. Our results demonstrate that Marlin effectively adapts to dynamic workloads, maintains high throughput, and sustains performance even when faced with Byzantine faults. Our experimental evaluation shows that Marlin swiftly adapts to changing workloads, consistently achieving high throughput despite the presence of Byzantine faults. The centralized architecture attains faster convergence and higher throughput due to coordinated shard management, while the decentralized architecture offers enhanced resilience by eliminating single points of failure. These results highlight the effectiveness of Marlin's adaptive sharding in balancing performance and fault tolerance in adversarial environments.

- **Chapter 2: Background** — Provides an overview of sharding strategies, trust assumptions, and the challenges of sharding in Byzantine environments.
- **Chapter 3: System Design** — Discusses the design of the Marlin system, including centralized and decentralized architectures and adaptive sharding mechanisms.
- **Chapter 4: Decentralized Architecture** — Details the decentralized architecture, including performance monitoring, proposal generation, and execution of resharding plans.
- **Chapter 5: Evaluation** — Presents experimental results, analyzing the system's performance, adaptability, and architectural impact.
- **Chapter 6: Related Work** — Examines prior research in adaptive distributed systems, Byzantine fault tolerance, and machine learning-based optimization.
- **Chapter 7: Remaining Tasks and Timeline** — Outlines future work, including correctness validation, workload generalization, and reproducibility efforts.
- **Chapter 8: Conclusion** — Summarizes the contributions of the dissertation, reflects on its broader implications, and suggests directions for future research.

CHAPTER 2

BACKGROUND

Sharding is a widely adopted technique for improving scalability in distributed systems DeCandia et al. (2007); Corbett et al. (2013). By partitioning data across multiple fault-tolerant clusters of nodes, systems manage large datasets efficiently, providing high throughput and low-latency operations. Sharding strategies are generally categorized into *static* and *dynamic* approaches.

2.1. Static and Dynamic Sharding

Static sharding shards data into fixed partitions using a predefined partitioning function. Systems like MySQL Cluster Corbett et al. (2013) and MongoDB Liu et al. (2012) implement static sharding to distribute data across nodes. This method is effective for stable workloads, as it minimizes coordination overhead and avoids the complexity associated with frequent rebalancing. 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 Quamar et al. (2013), Schism Curino et al. (2010), Spanner Corbett et al. (2013), and CosmosDB Frey and Goes implement dynamic sharding to adaptively repartition data based on real-time workload statistics. This adaptability improves resource utilization and system responsiveness, particularly in environments where workload patterns fluctuate. Recent dynamic sharding frameworks, such as those proposed by Gupta et al. (2021) and Haroon et al. (2024), integrate machine learning to adaptively partition workloads and predict workload shifts, optimizing data distribution in real-time. These methodologies leverage gradient elimination and predictive modeling to enhance shard adaptability in adversarial conditions, paving the way for improved scalability and fault tolerance.

2.2. Trust Assumptions in Sharding

Most existing sharding solutions, especially those employing dynamic sharding Quamar et al. (2013); Curino et al. (2010), assume a **trusted environment**, where all nodes behave reliably without malicious intent and the failure model of nodes is fail-stop (i.e., non-Byzantine). In these settings, the primary focus is on performance and scalability, without the necessity to mitigate adversarial threats. This assumption simplifies the design and implementation of sharding mechanisms but overlooks the challenges posed by untrusted environments where nodes may exhibit Byzantine faults.

Untrusted environments, in contrast, follow the Byzantine failure model, where nodes may behave arbitrarily or maliciously Lamport et al. (1982). Ensuring system reliability and consistency in such scenarios requires implementing Byzantine Fault Tolerance (BFT) protocols, such as Practical Byzantine Fault Tolerance (PBFT) Castro and Liskov (1999a). Sharding in untrusted environments is more complex due to the need for secure coordination across untrustworthy shards.

2.3. Static Sharding in Untrusted Environments

Untrusted environments mainly rely on BFT protocols to secure data and operations, as malicious nodes can interfere with shard adjustments and cross-shard 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 Amiri et al. (2021); Sousa and Bessani (2015). However, static sharding mechanisms lack adaptability to changing workloads, potentially leading to load imbalances and performance degradation over time.

Sharded data management systems must process both *intra-shard* and *cross-shard* transactions. While intra-shard transactions can be processed using BFT protocols (e.g., PBFT Castro and Liskov (1999a)), cross-shard transactions are more complex, as multiple shards are involved in processing. Cross-shard transactions can be managed in:

- **A centralized manner (coordinator-based):** A single coordinator orchestrates transaction commitment across shards using an atomic commitment protocol such as two-phase commit (2PC). Examples include AHL Dang et al. (2019), Saguaro Amiri et al. (2023), and Blockplane Nawab and Sadoghi (2019).

- **A decentralized (flattened) approach:** Each shard participates in the commitment protocol without a single central coordinator, as in SharPer Amiri et al. (2021).

We mainly 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 untrustworthy shards. In contrast to the 2PC protocol where a single coordinator manages the commitment of each cross-shard transaction, the coordinator itself needs to be implemented as a Byzantine fault-tolerant cluster.

Figure 2.2 presents a coordinator-based cross-shard protocol across two clusters A and B where cluster A plays the role of coordinator. As can be seen, the leader of the coordinator cluster initiates the protocol by locally processing the incoming request (using PBFT) and multicasting a prepare message, including the request, to the involved clusters (i.e., B). Each shard then independently reaches agreement on the order of the request internally and sends prepared message to the coordinator cluster. Upon receiving sufficient prepared messages, an instance of PBFT will be initiated among nodes of the coordinator cluster to decide whether to commit or abort the transaction and all other clusters become aware of the decision. Reaching local agreement in each phase of the protocol is needed to prevent a malicious leader from sending invalid messages to other clusters.

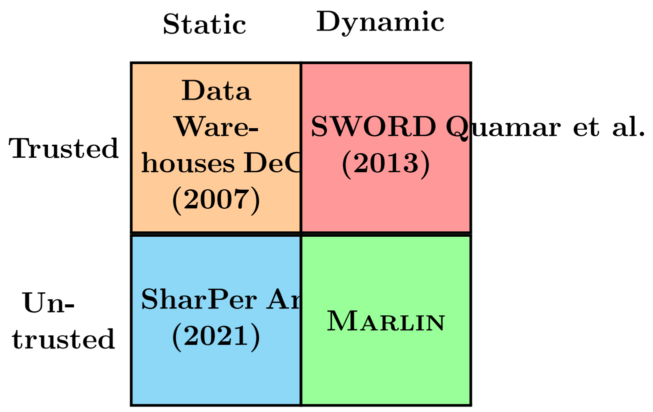

Figure 2.1: Classification of sharding strategies based on trust assumptions and sharding dynamics.

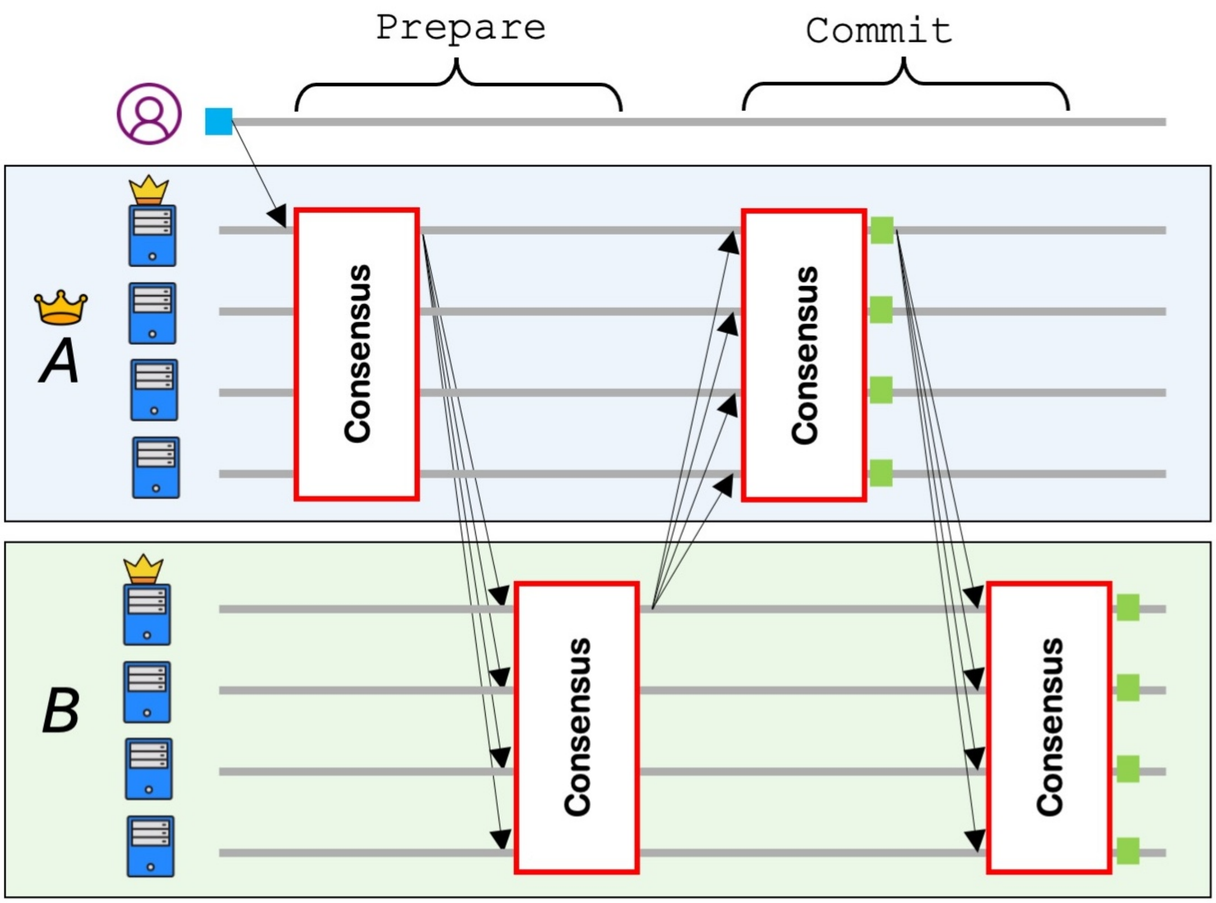

Figure 2.2: Byzantine Fault-Tolerant Two-Phase Commit Protocol for Cross-Shard Transactions.

2.4. Dynamic Sharding

Despite the need for adaptability in untrusted environments, existing dynamic sharding solutions have largely assumed a trusted environment, where nodes cooperate reliably and failures follow the fail-stop model Quamar et al. (2013); Curino et al. (2010). Such an assumption leaves systems vulnerable to more severe faults and attacks in environments where nodes may act maliciously and compromise data integrity and availability.

Integrating dynamic sharding with BFT protocols, however, presents significant challenges. Dynamic sharding requires frequent adjustments to shard boundaries, which, in an untrusted environment, necessitates secure coordination and consensus across potentially malicious nodes. The additional communication overhead and complexity of achieving consensus in BFT systems make it difficult to adapt shard boundaries efficiently. There is a clear need for a system that dynamically adapts to workload changes while providing Byzantine fault tolerance. Existing solutions primarily focus on either static sharding in the presence of Byzantine failures, which lacks adaptability, or provide dynamic sharding in trusted environments where there is no Byzantine failure. This gap leaves distributed data management systems susceptible to adversarial risks when deployed in untrusted contexts.

2.4.1. Towards Dynamic Sharding in Untrusted Environments

Recent work has begun addressing dynamic sharding under adversarial conditions. Guerraoui et al. (2024) and Gupta et al. (2020) propose Byzantine-resilient optimization methods and fault-tolerant SGD techniques, focusing primarily on theoretical resilience. Building on these concepts, **Marlin** offers a more practical implementation that combines dynamic adaptability with fault tolerance. Experiments show a 33% reduction in cross-shard transaction rates compared to static approaches, illustrating its effectiveness in real-world workloads.

This work introduces MARLIN, a distributed data management system that supports dynamic sharding in untrusted environments. MARLIN integrates adaptive sharding mechanisms with atomic commitment and consensus protocols capable of tolerating Byzantine failures, thus providing adaptivity, scalability, and security. Figure 2.1 categorizes sharding strategies based on trust assumptions and sharding dynamics, positioning MARLIN alongside existing systems and showcasing the need for fully dynamic and Byzantine fault-tolerant sharding solutions.

Marlin (and likewise MARLIN) bridges the gap between static and dynamic systems by providing:

- Fully adaptive mechanisms to handle workload variability.
- Robust fault tolerance through dual-layer PBFT consensus.
- Scalability in heterogeneous and adversarial workloads.

Additionally, novel approaches like Byzantine machine learning Guerraoui et al. (2024) and fault-tolerant SGD methods Gupta et al. (2020) serve as complementary solutions for enhancing system resilience and scalability. These contributions align closely with Marlin's and MARLIN 's focus on unifying adaptive sharding with robust Byzantine fault tolerance.

CHAPTER 3

SYSTEM DESIGN

In this chapter, we present Marlin, a distributed database system designed to address the challenges of dynamic sharding in untrusted environments. We begin by defining the system model and assumptions and then introduce a centralized architecture for the proposed system. Marlin 's architecture combines adaptive sharding with Byzantine fault tolerance to address challenges in distributed systems.

3.1. System Model and Data Organization

Marlin operates in a distributed system consisting of a network of C clusters. Each cluster C_i contains $n = 3f_i + 1$ nodes, where f_i is the maximum number of Byzantine faulty nodes the cluster can tolerate concurrently Lamport et al. (1982). The network is assumed to be partially synchronous: message delays are unbounded but messages are eventually delivered. Temporary network partitions or delays may occur but will eventually resolve, allowing the system to make progress.

Data in Marlin is partitioned into *shards*, with each shard managed by one of the C clusters. Sharding enhances scalability by distributing data and workloads across multiple clusters. Each shard is replicated across all n nodes within its respective cluster. To tolerate Byzantine failures, Marlin employs two agreement protocols: a *Byzantine Fault-Tolerant (BFT)* consensus protocol for intra-cluster operations and a *Byzantine Fault-Tolerant Atomic Commitment* protocol for coordinating cross-shard transactions.

For intra-cluster consensus, we use the Practical Byzantine Fault Tolerance (PBFT) Castro and Liskov (1999a) protocol to agree on the execution order of transactions and state changes. For transactions involving multiple shards, Marlin employs a Two-Phase Commit protocol integrated with BFT

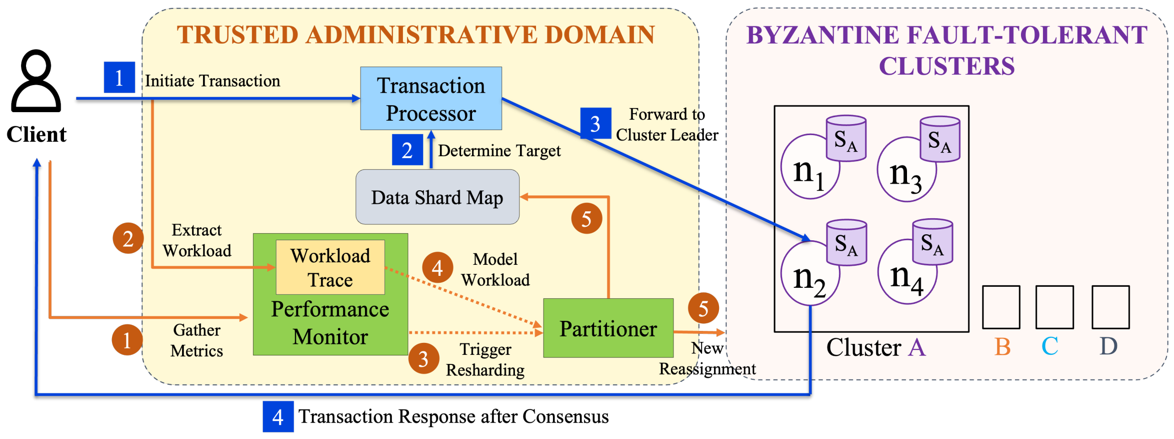

Figure 3.1: Centralized Architecture.

consensus at each phase (BFT-2PC), ensuring atomicity and consistency across shards even if nodes become faulty during the transaction process. We detail this protocol in Section 2.3 (see Figure 2.2).

3.2. Centralized Architecture

The centralized architecture of Marlin serves as a baseline model for distributed databases in untrusted environments. As shown in Figure 3.1, the replicas are organized into clusters, each managed by a cluster leader. 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-cluster transactions are executed correctly, even in the presence of up to f faulty nodes. However, the central coordinator itself is not replicated and thus constitutes a single point of failure.

When the central coordinator receives a client request (1 in Figure 3.1), 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 node of the corresponding cluster (3). If more than one shard is involved in the transaction (i.e., a cross-shard transaction), the transaction will be forwarded to the leader of one of the involved clusters.

Upon receiving the transactions, the cluster leader (node n_2 in our example) initiates a PBFT consensus instance to reach an agreement on the order of the transaction. Depending on the transaction type, the leader either runs PBFT within its cluster for intra-cluster transactions or employs the BFT-2PC protocol for cross-cluster transactions, as described in Section 2.3. Once the transaction is executed, the response is sent back to the client by all nodes following the PBFT protocol (4).

3.3. Monitoring and Shard Management

The *Performance Monitor* assesses system performance by collecting metrics such as throughput and latency (1 in Figure 3.1). 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*—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 monitor initiates the resharding process (2). Performance degradation is quantified based on 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, and hyperedges represent transactions that access those data items.

The Partitioner uses hypergraph partitioning algorithms, specifically, KaHyPar Schlag et al. (2023), to partition the hypergraph into k balanced partitions (shards) while minimizing the number of cross-shard transactions (the hyperedge cut). After partitioning, the Data Shard Map is updated with the new reassignments (5), which are also shared with the other clusters.

3.4. Hypergraph Partitioning

Specifically, the partitioner solves the following optimization problem:

$$\min_{\mathcal{P}} \quad \mathrm{cut}(H, \mathcal{P}) \tag{3.1}$$

$$\text{subject to} \quad \forall P_i \in \mathcal{P},\ |V_{P_i}| \leq (1+\epsilon)\frac{|V|}{k} \tag{3.2}$$

where:

- $H(V, E)$ is the hypergraph representing the workload,
- $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ is a partition of the vertex set V into k shards,
- $\mathrm{cut}(H, \mathcal{P})$ denotes the total weight of hyperedges cut by the partition $\mathcal{P}$, and
- ϵ represents the imbalance tolerance factor.

KaHyPar is chosen for its efficiency in handling large-scale hypergraphs, achieving near-linear time complexity in practical scenarios Schlag et al. (2023). 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 Kuhn (1955). Assuming that:

- $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ were the original shards
- $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ are the new shards
- $cost(S_i, P_j)$ denotes the cost of moving data from S_i to P_j

It solves the assignment problem of mapping each shard in $\mathcal{S}$ to exactly one shard in $\mathcal{P}$ while minimizing the total cost of data transfer.

3.5. Limitations

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

First, the central coordinator constitutes a single point of failure and 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. One could address this limitation by replicating the coordinator on multiple nodes where a cluster of nodes (i.e., a coordinator committee). However, this would introduce additional complexity and overhead in a few ways. First, the nodes in the coordinator committee must agree on the same global view of the system, which requires running BFT consensus protocol among them, and second, the data nodes must communicate with all coordinator replicas and maintain consensus on top of the existing intra-cluster consensus, further complicating the system design.

Second, the coordinator-based solution, whether utilizing a single coordinator 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.

CHAPTER 4

DECENTRALIZED ARCHITECTURE

To address the limitations of centralized architecture, Marlin adopts a *decentralized model* where shard management and decision-making are distributed across nodes to enhance resilience against Byzantine failures:

4.1. Architecture Overview

In the decentralized architecture (as shown in Figure 4.1), nodes communicate directly in a peer-to-peer network. Each node maintains its own copy of the *Data Shard Map* and a *Decentralized Manager* that stores workload traces and performance metrics, generating proposals to facilitate resharding. Nodes make autonomous decisions based on local observations of workload and performance, collaborating with others to reduce cross-shard transactions and improve system performance while maintaining load balance.

Each client also maintains a local Data Shard Map to route transactions. If a client's Data Shard Map is outdated due to data movement, the cluster receiving the request forwards it to the correct cluster based on its own Data Shard Map. Upon receiving the response, the client updates its Data Shard Map accordingly.

To initiate a transaction, as presented in Figure 4.1, a client first checks its local Data Shard Map to determine the cluster containing the required data and sends the transaction to the leader of the cluster (1). Upon receiving the request, the *Transaction Router* component of the leader parses the transaction and checks its local Data Shard Map to identify the type of transaction (2). If the data resides within the same cluster, the transaction executes locally using PBFT. If not, the request is routed to the leader of all other involved clusters (3).

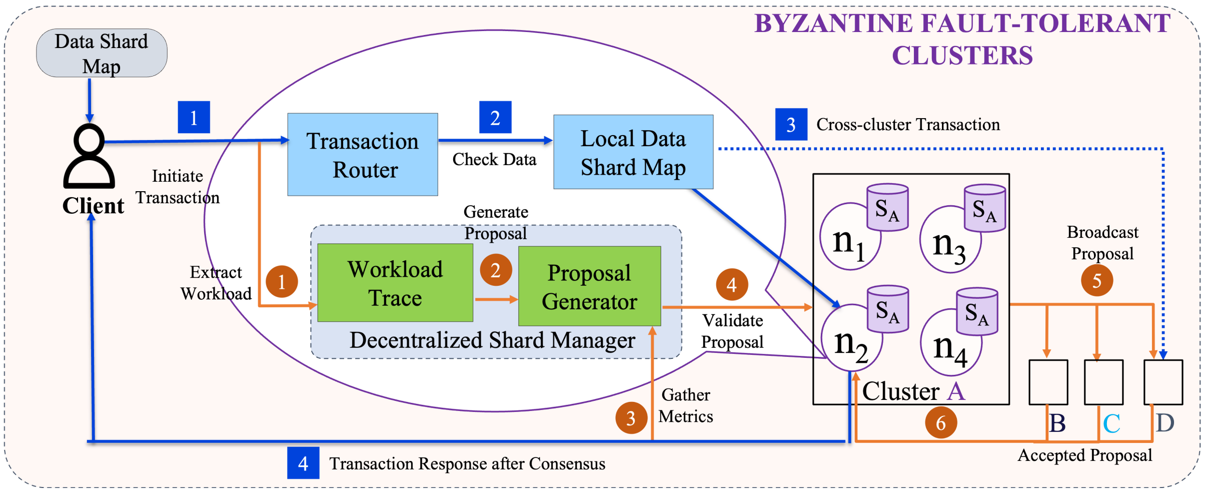

Figure 4.1: Decentralized Architecture.

Intra-shard transactions use PBFT and cross-shard transactions use two-phase commit (on top of PBFT) to reach agreement on the order of transactions (described in Section 2.3). Once a transaction is executed, nodes send reply messages to the client (4). If the client's Data Shard Map is outdated, the client updates the map based on the received replies.

4.2. Global Convergence with Partial View of Hash Table

To address our problem where nodes have a partial view of the hash table and must achieve global convergence in a decentralized setting, implement a strategy inspired by distance-vector routing:

1. Goals

- Achieve a consistent, global view of the hash table from partial views.
- Minimize communication overhead and convergence time.
- Ensure performance stability in the decentralized system.

2. Key Strategies

- **Hash Table Updates**: Each node exchanges partial hash table views and updates its own based on neighbors' information.

- **Split Horizon**: Prevent nodes from re-propagating updates back to the source to avoid loops.
- **Triggered Updates**: Propagate changes immediately to reduce convergence time.
- **Convergence Criteria**: Define a condition for stopping updates when the global view stabilizes.

In code:

- Adding convergence detection using the `hasConverged` function.
- Improving update propagation with the `mergeShardMappings` function.
- Triggering updates only when necessary based on shard reassignments.

These address - unnecessary propagation of updates, long convergence times, and incorrect shard mappings.

Key Changes and Their Effects

1. Convergence Detection (`hasConverged`)

$$\text{Converged} = \forall k \in \text{Transactions}, \quad \text{CS}(t+1) = \text{CS}(t), \tag{4.1}$$

where:

- $\text{CS}(t)$ is the cross-shard percentage at time t.
- k is a transaction.

This ensures that updates are halted when the cross-shard percentage stabilizes, reducing unnecessary communication and improving system efficiency.

2. Merging Shard Mappings (`mergeShardMappings`)

The `mergeShardMappings` function combines local and incoming shard mappings, updating only when discrepancies are detected- the client sends you a dataitem which is not present in your shard.

The update rule is:

$$\text{Updated} = \exists k, \quad \text{LocalMap}(k) \neq \text{IncomingMap}(k), \tag{4.2}$$

where:

- $\text{LocalMap}(k)$ is the current mapping for key k.
- $\text{IncomingMap}(k)$ is the mapping received from a neighbor.

This function prevents redundant updates and ensures only meaningful changes are propagated.

3. Triggered Updates

The propagation of updates is now conditional upon the number of shard reassignments:

$$\text{Propagate} = \text{NumReassignments} > 0. \tag{4.3}$$

This reduces the overhead caused by propagating updates when no significant changes have occurred.

How above Changes Solve These Problems:

- **Convergence Detection**: By halting updates once shard mappings stabilize, unnecessary communication is avoided.
- **Efficient Propagation**: The `mergeShardMappings` function ensures only meaningful changes are propagated.
- **Optimized Triggering**: Updates are propagated only when reassignments occur, reducing communication overhead.

The integration of `hasConverged`, `mergeShardMappings`, and triggered updates significantly improves the efficiency and correctness of the decentralized resharding algorithm. These changes address key inefficiencies and ensure the system converges faster with minimal communication overhead.

4.3. Performance Monitoring

Each node continuously monitors its local performance metrics, such as throughput and latency. When significant performance degradation beyond predefined thresholds is detected, the node initiates the resharding process.

Thresholds are configurable parameters set by system administrators based on historical data, workload characteristics, or service-level agreements (SLAs). For example, resharding might be considered if latency increases by more than 30% or throughput drops below a certain rate.

These thresholds are dynamically updated by analyzing trends in local workload and performance, preventing unnecessary resharding due to transient fluctuations and ensuring responsiveness to significant performance changes. If a node observes consistent changes in workload patterns, it may adjust the thresholds to better reflect the new operating conditions. This dynamic adjustment helps prevent unnecessary resharding due to transient fluctuations and ensures that the system remains responsive to significant performance changes. The resharding process is summarized in pseudocode in Algorithm 1

4.4. Localized Heuristic Resharding Mechanism

We introduce the *Localized Heuristic Resharding (LHR)* algorithm, enabling nodes to independently adjust shard boundaries based on local workload observations and heuristic strategies. By dynamically adjusting data placement, LHR minimizes cross-shard transactions and eliminates the need for a central coordinator, reducing bottlenecks and improving scalability.

We use the following simple example throughout this section to illustrate LHR algorithm. The system includes three shards and a set of 8 transactions, as shown in Table A.1. We assume that initially, shard S_1 maintains keys A, E, and F, shard S_2 maintains keys B and C, and shard S_3 maintains keys D and G. We intentionally consider a workload with a high percentage of cross-shard transactions to demonstrate the resharding process.

ID	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Keys	A, B	C, D	C, D	E	A, B	B, A	E, F	E, G

Table 4.1: Transactions and their accessed keys

4.5. Proposal Generation

Once a node detects performance degradation, it generates a *resharding proposal* using LHR to optimize data placement and reduce cross-shard transactions. The steps involved in generating and validating the proposal are shown in Figure 4.1:

① Extracting the workload: The node extracts workload information from local transaction logs to analyze recent transaction patterns. This helps identify high-frequency transactions and key data access patterns.

② Proposal generation: Using the extracted workload data, the node generates a resharding proposal. Transactions are sorted by frequency within a specified time interval, prioritizing costly transactions. Data items involved in each transaction are assigned to shards that access them most frequently, co-locating frequently accessed data items.

③ Gathering metrics: The node evaluates the impact of the proposed data assignments on local performance metrics. This ensures that the proposed changes align with system goals for reducing cross-shard communication and balancing loads.

④ Proposal validation: The finalized resharding proposal is sent to all replicas within the initiating cluster. Each replica validates the proposal by simulating its impact on shard balance and cross-shard transaction rates. Once a quorum of signatures is collected, the proposal is deemed validated.

⑤ Broadcast to all involved clusters: The validated proposal is sent to the leaders of all involved clusters (e.g., Clusters B, C, and D in Figure 4.1). Each receiving cluster evaluates the proposal independently to assess its impact on shard balance and cross-shard transactions.

⑥ Accepted proposal execution: If the proposal satisfies acceptance criteria (e.g., reduced cross-shard transactions, balanced shard loads), nodes in the receiving clusters sign the proposal. The leader consolidates feedback from its cluster and sends the accepted proposal back to the leader of the initiating cluster.

Continuing with our example, based on the keys, each node in cluster S_1 records transactions T_1, T_4, T_5, T_6, T_7 and T_8 as they have one or more keys from the set of key maintained by S_1, i.e., A, E and F. Transaction T_4 gets filtered since it only involves one key. The remaining transactions are sorted based on the frequency of accessed key combinations, as shown in Table 4.2. Based on the frequency table, the leader of cluster S_1 proposes moving key A to cluster S_2 to co-locate frequently accessed pairs. It skips moving E since it parses transaction T_7 with accessed keys E and F first and decides to keep both keys in S_1. In the beginning, shard S_1 maintains keys A, E, and F, shard S_2 maintains keys B and C, and shard S_3 maintains keys D and G. After co-locating A and B, shard S_1 maintains keys E and F, shard S_2 maintains keys A, B and C, and shard S_3 maintains keys D and G.

Accessed Keys	**Frequency**	**Shards Involved**
{A,B}	3	S1, S2
{E,F}	1	S1
{E,G}	1	S1, S3

Table 4.2: Transactions sorted by frequency

4.6. Proposal Validation

Before sending the proposal to other clusters, the initiating node broadcasts it to all replicas within its cluster. Each replica independently validates the proposal by simulating its potential impact on performance metrics such as cross-shard transaction rates and shard balance, using local state snapshots. Snapshots can be captured either based on transaction frequency or at regular time intervals.

Marlin adopts time-based snapshots as they are decoupled from transaction execution and provide a lightweight mechanism for maintaining approximate state consistency during validation. While time-based snapshots may diverge slightly due to timing differences, they remain sufficient for validation purposes Bernstein and Goodman (1981). For stricter consistency, capturing snapshots based on transaction sequence numbers could be used but would introduce higher system overhead.

Each replica validates the proposal using its local snapshot, simulating its impact on workload metrics. If the proposal is valid, the replica signs it with a cryptographic signature and sends the signature back to the initiating node. Once the initiating node collects signatures from at least $2f + 1$ replicas (where f is the maximum number of tolerated Byzantine faults), the proposal is considered valid and forwarded to the affected clusters.

If a malicious node attempts to avoid submitting a valid proposal or disseminates invalid ones, it will fail to gather the required $2f + 1$ signatures, preventing acceptance. Since proposals at this stage do not directly alter system state but serve as inputs for the PBFT-based execution phase, any invalid or maliciously approved proposals will be identified and rejected during consensus, ensuring the integrity of state-altering operations.

Using cryptographic signatures for proposal validation provides a lightweight agreement mechanism without the overhead of running PBFT during this phase. Signatures ensure authenticity and integrity, and the quorum of $2f + 1$ guarantees progress even with malicious or unresponsive nodes. If the leader is malicious, PBFT's view-change mechanism triggers a consensus process among remaining nodes, preserving system reliability. This can be formalized as the following optimization problem:

$$\begin{aligned} \min_{\mathcal{M}} \quad & \frac{1}{|T|} \sum_{t \in T} \delta(t, \mathcal{P}_{\text{new}}) \\ \text{subject to} \quad & \frac{1}{|T|} \sum_{t \in T} \delta(t, \mathcal{P}_{\text{new}}) < \frac{1}{|T|} \sum_{t \in T} \delta(t, \mathcal{P}_{\text{current}}) \\ & S_{\min} \leq |V_{P_i}| \leq S_{\max}, \quad \forall P_i \in \mathcal{P}_{\text{new}} \end{aligned}$$

where:

- $\frac{1}{|T|} \sum_{t \in T} \delta(t, \mathcal{P})$: Average cross-shard transaction rate under shard mapping $\mathcal{P}$.
- $\delta(t, \mathcal{P})$: Indicator function that returns 1 if transaction t is a cross-shard transaction under mapping $\mathcal{P}$, and 0 otherwise.
- T: Set of all transactions.

- $S_{\min}$ and $S_{\max}$: Minimum and maximum shard size limits.
- V_{P_i}: Set of data items assigned to shard P_i.

In our example, the leader of cluster S_1 then broadcasts the resharding proposal to other nodes in its cluster. Each backup node in cluster S_1 validates the resharding proposal by (1) fetching transactions T_1, T_4, T_5, T_6, T_7, and T_8, (2) computing the new percentage of cross-shard transactions (A and B are co-located and T_8 is the only remaining cross-shard transaction, thus the number of cross-shard transactions reduces from 4 to 1), and (3) validate the proposal by signing it.

4.7. Proposal Evaluation

When a receiving cluster receives a signed proposal, each node S_r in the cluster evaluates the proposed data movements. To perform the evaluation, the cluster first simulates applying the proposed data movements $\mathcal{M}_{S_r}$ to its local Data Shard Map. The cluster then evaluates batches of proposed data movements, checking each batch's impact on cross-shard transaction rates and shard load balance. If a batch reduces cross-shard transactions without causing imbalance, the cluster accepts those data items. Imbalance is defined by the shard size thresholds $S_{\min}$ and $S_{\max}$. The receiving cluster's decision-making process is formalized as the following optimization problem:

$$\begin{aligned} \max_{\mathcal{M}'_{S_r} \subseteq \mathcal{M}_{S_r}} \quad & \Delta C_{S_r} = \frac{1}{|T|} \left(\sum_{t \in T} \delta(t, \mathcal{P}_{\text{current}, S_r}) - \sum_{t \in T} \delta(t, \mathcal{P}_{\text{new}, S_r}) \right) \\ \text{subject to} \quad & \Delta C_{S_r} > 0 \\ & S_{\min} \leq \left| V_{S_r} \cup K_{\mathcal{M}'_{S_r}} \right| \leq S_{\max} \end{aligned}$$

where:

- ΔC_{S_r}: Reduction in cross-shard transaction rate for shard S_r.
- $K_{\mathcal{M}'_{S_r}}$: Subset of data items proposed for reassignment to S_r.
- V_{S_r}: Set of data items currently assigned to shard S_r.

If the proposal meets the acceptance criteria, nodes in the receiving cluster sign it. When the cluster leader receives signatures from at least $2f + 1$ nodes, it confirms the newly negotiated proposal and sends it back to the initiating cluster. If the leader is malicious, the PBFT view-change mechanism ensures its replacement by triggering a consensus process among the remaining nodes. This mechanism ensures the continuity of operations even in the presence of a faulty leader, preserving the system's Byzantine fault tolerance.

Continuing with our example and upon receiving the proposal from cluster S_1, cluster S_2 evaluates moving key A to its shard. It loads the transactions it received from its snapshot, T_1, T_2, T_3, T_5 and T_6. Since moving key A will reduce the number of cross-shard transactions in cluster S_2 from 5 to 2 without overloading S_2, cluster S_2 accepts the proposal and signs it. After receiving signatures from the required number of nodes, the initiating node finalizes and executes the resharding plan. This new distribution reduces global cross-shard transactions from 6 to 3, optimizing performance while maintaining balanced shard sizes. The number of cross-shard transactions will further reduce as other shards also repeat the resharding process.

4.8. Executing the Resharding Plan

The initiating node aggregates the responses, forming a final resharding plan $\mathcal{M}_{\text{final}}$. The node then uses the two-phase commit protocol to execute the plan across affected shards. After execution, all nodes update their local Data Shard Maps to reflect the new data distribution. During the two-phase commit process, failures such as node crashes or Byzantine behavior are managed using PBFT's fault-tolerance guarantees. If a node fails to respond or provides inconsistent responses, the protocol ensures that the operation either proceeds with a valid quorum or is safely aborted, avoiding partial state changes.

The initiating node aggregates the responses, forming a final resharding plan. The node then uses the two-phase commit protocol to execute the plan across affected shards. After execution, all nodes update their local Data Shard Maps to reflect the new data distribution. During the two-phase commit process, failures such as node crashes or Byzantine behavior are managed using PBFT's fault-tolerance guarantees. If a node fails to respond or provides inconsistent responses,

the protocol ensures that the operation either proceeds with a valid quorum or is safely aborted, avoiding partial state changes.

This distributed workflow ensures that decisions are made collaboratively and consistently across the system. Resharding proposals are validated by cluster replicas, which assess their impact on metrics like cross-shard transaction rates and shard balance. Only proposals that meet these criteria are signed and forwarded for execution, ensuring that changes are beneficial and maintain consistency.

By combining localized heuristic decisions with collaborative validation and robust execution protocols, MARLIN achieves dynamic adaptability to workload changes while maintaining data consistency and system reliability. This design ensures that the system behaves predictably, even in the presence of Byzantine faults, without requiring centralized coordination.

Algorithm 1 Decentralized Resharding Workflow

```
Require: LocalMetrics, ShardState
Ensure: Shard redistribution if accepted by consensus
  procedure GeneratePropoSAL(LocalMetrics, ShardState)
      if LocalMetrics.latency > LATENCY_THRESHOLD then
          Proposal ← CreateReshardingPlan(ShardState, LocalMetrics)
          Broadcast(Proposal, relevant nodes)
  procedure ValidateProposal(Proposal, LocalState)
      if Proposal is valid against LocalState then
          Sign(Proposal)
          SendSigned(Proposal, proposing node)
      else
          Reject(Proposal)
```

CHAPTER 5

EVALUATION

In this chapter, we evaluate both the centralized and decentralized architectures of Marlin to assess their performance and adaptability in Byzantine environments. We do so through a series of experiments designed to answer three main research questions:

1. **RQ1. Performance:** How efficient are the different architectures of Marlin in terms of throughput?

2. **RQ2. Adaptability:** How efficiently does Marlin adapt to changes in workloads over time?

3. **RQ3. Architectural Impact:** How does the choice of Marlin architecture and partitioning strategy impact performance?

5.1. Experimental Setup

Our experiments leverage a custom workload generator that models a key-value store with N = 10,000 records. These records are partitioned across four clusters using range partitioning, with 2,500 records assigned to each cluster. Each cluster's records are fully replicated across four nodes, enabling tolerance for $f = 1$ Byzantine fault per cluster. Transactions are categorized into intra-shard transactions, confined to a single shard, and cross-shard transactions, which involve records spanning multiple shards. The default workload comprises 40% cross-shard transactions and 60% intra-shard transactions. This distribution is chosen to simulate contention-heavy environments, such as social networks or e-commerce systems. The 40% ratio, while higher than the commonly used 10-20%, provides a rigorous test of the system's ability to handle cross-shard coordination. The percentage is reset at each workload phase, ensuring consistent evaluation across configurations. Workload skewness is introduced to reflect realistic data access patterns, where 70% of transactions target a small subset of records, creating data hotspots.

Workload Features In summary, our workload features the following characteristics:

- *Key-Value Data Layout:* Keys are distributed using a range-partitioning scheme across shards. This setup is initialized through a Redis-backed shard-to-data mapping.

- *Transaction Mix:* Transactions are generated with a fixed ratio of cross-shard to intra-shard operations. Cross-shard transactions involve at least two randomly selected shards, while intra-shard transactions select records confined to the initiating shard.

- *Hotspot Configuration:* Skewness is introduced by sampling keys from a 70% bias pool for a subset of frequently accessed records, ensuring realistic contention.

Configurations. We compare MARLIN's centralized and decentralized architectures with a no-resharding configuration to evaluate the performance of different configurations. The centralized configuration employs a global coordinator to optimize data placement and minimize cross-shard transactions using a hypergraph partitioning strategy. The decentralized configuration relies on local information to make resharding decisions via localized heuristics, while the no-resharding configuration maintains the initial data placement without any adaptive resharding.

Infrastructure. We implement MARLIN in Python and deploy it on 8 physical machines. Each machine has an Intel Xeon Platinum 8253 CPU with 16 cores each with 2 threads. Additionally, each machine is configured with 128 GB of DDR4 memory and 1 TB of SSD storage, connected via a 10 Gbps Ethernet network. The operating system used is Ubuntu 20.04 LTS.

In our experiments, by default we configure four clusters, each with four nodes designed to tolerate up to $f = 1$ Byzantine faults, resulting in a total of 16 nodes across the system. We assigned specific logical processors (cores) to each node, allocating 4 logical processors to each node. This ensured dedicated processing resources and prevented resource contention between nodes and clusters.

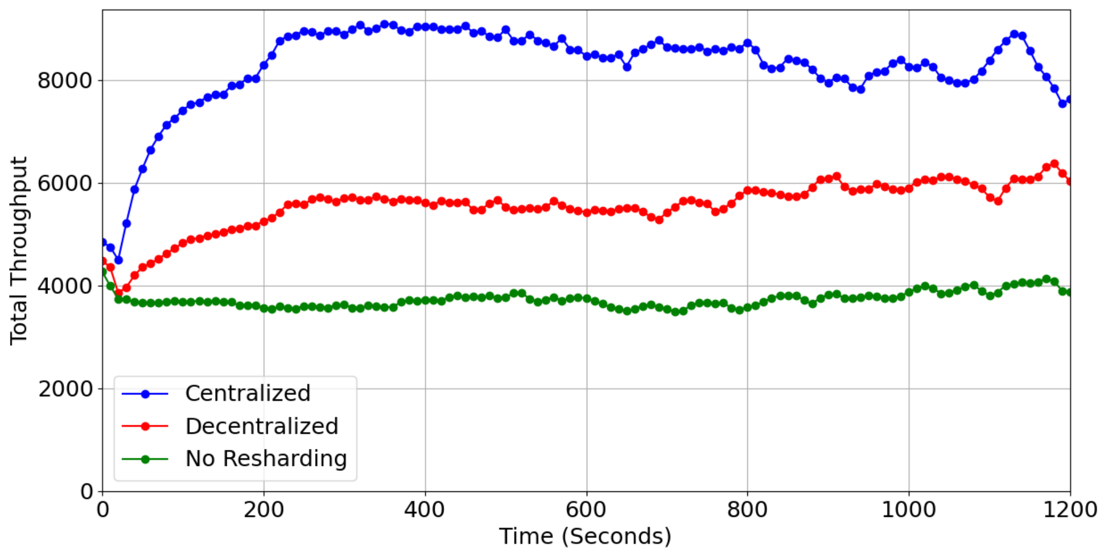

Figure 5.1: Throughputs of centralized, decentralized, and no resharding configurations under a fixed workload.

5.2. RQ1. Performance.

Setup. We first evaluate the performance of the three configurations of Marlin under a fixed workload to assess the effectiveness of the adaptive resharding mechanism in minimizing cross-shard transactions and improving throughput. We do so by comparing the throughputs of each configuration over time focusing on the convergence time and the final throughput achieved. As mentioned in the setup, each configuration considers four clusters, each with four nodes ($f = 1$, faulty node per cluster), and clients generating a fixed workload with 40% cross-shard transactions, each involving two shards on average. We ran the experiment for 10 minutes, sufficient for the system to converge to a steady state. The results are shown in Figure 5.1, where each curve represents the throughput over time for the centralized, decentralized, and no resharding configurations. We plot the simple moving average of the throughput in each case over a 5-second window to ignore short-term fluctuations and focus on the overall trend.

Results. Initially, all configurations start with similar low throughput (around 4000 transactions per second) due to the high percentage of cross-shard transactions. With resharding, the centralized architecture rapidly achieves a throughput of around 8000 transactions per second within 170 seconds and further increases to around 8500 transactions per second within 320 seconds. The decentralized architecture stabilizes at around 6000 transactions per second after 370 seconds. The configuration without resharding maintains the same throughput throughout the experiment.

We observe that the centralized architecture adapts the fastest and achieves higher throughput as compared to the decentralized architecture. From the experiments we conduct for **RQ3** in Section 5.4, we attribute this difference mainly to the partitioning schemes employed by each architecture. Specifically, the centralized architecture employs a hypergraph-based partitioning algorithm that globally optimizes data placement based on observed transaction patterns as discussed in Section 3.2. This also allows the centralized architecture to achieve higher throughput in its optimal state for the workload.

In contrast, the decentralized architecture requires more time to converge because each cluster only has local information and lacks a global system view. As discussed in Section 4, the clusters need to communicate with each other to exchange information about the shards they contain due to the absence of a central coordinator. The decentralized architecture also moves data in small increments to avoid network congestion. While more fault-tolerant due to the lack of a single point of failure, the decentralized approach may not find the globally optimal partitioning, leading to slightly lower throughput. We explore these differences further in Section 5.4. We observe a throughput improvement of approximately 2.1x for the centralized architecture and 1.5x for the decentralized architecture after resharding.

5.3. RQ2. Adaptability.

Setup To evaluate MARLIN's ability to adapt to changing workloads, we simulate scenarios where transaction patterns change periodically. This experiment assesses the system's performance in dynamic conditions by adjusting shard allocations in response to workload shifts.

We ran the system for 36 minutes, switching between three distinct workloads: workloads A, B, and C, every 700 seconds. We start with workload A, which has around 40% cross-shard transactions.

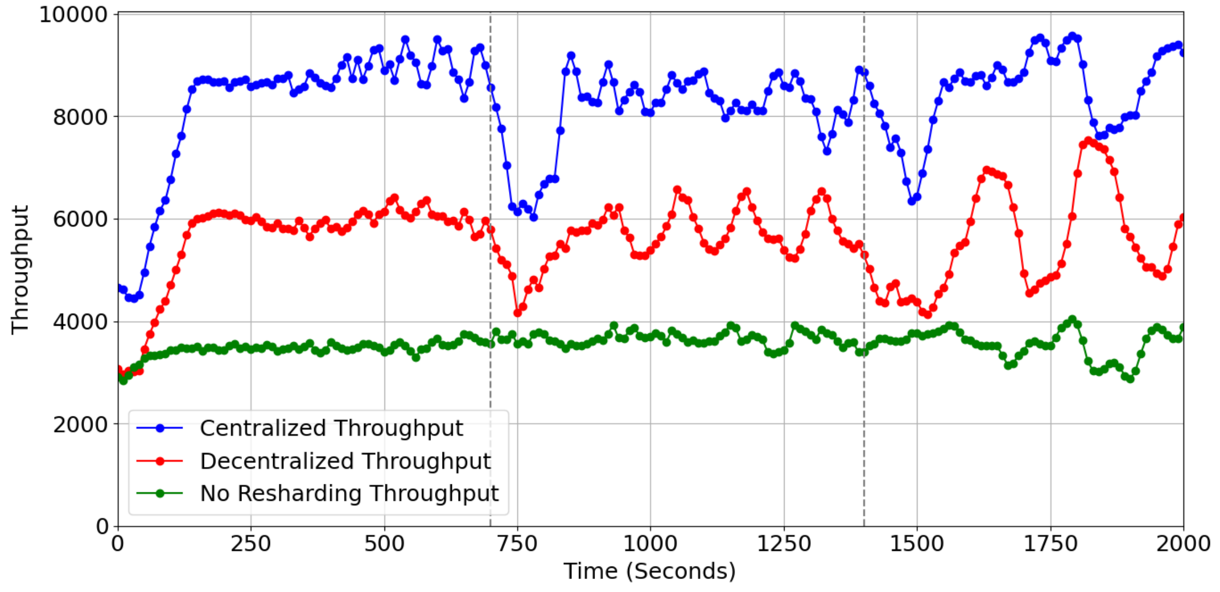

Figure 5.2: Throughput over time under dynamic workloads for centralized, decentralized, and no-resharding configurations. Vertical dashed lines indicate workload changes.

The centralized and decentralized systems then reshard this data to adapt to this workload. After 700 seconds, we switch to workload B containing around 40% cross-shard transactions with respect to the distribution of the resharded data. As before, both systems again reshard the data to adapt to workload B. After another 700 seconds, this process is repeated when we switch to workload C, again with around 40% cross-shard transactions with respect to the resharded data distribution. This approach allows us to rigorously test Marlin's adaptability to adapt to dynamic workload conditions and evaluate whether the system can maintain high performance even as transaction patterns fluctuate.

We compare the centralized and decentralized architectures of Marlin against a baseline configuration without adaptive resharding (referred to as *No-Resharding*). All configurations use the default parameters unless specified otherwise. The performance is monitored over time across the changing workloads.

Results. Figure 5.2 shows the throughput over time for each configuration. Like in the previous experiment, we plot the simple moving average of the throughput over time with a 5-second window for each configuration.

Initially, all configurations have low throughput due to the high percentage of cross-shard transactions in Workload A. Upon detecting a significant performance degradation, a 33% drop in throughput compared to their previous optimal states, the centralized and decentralized architectures trigger resharding. This threshold ensures timely adaptation to changes in workload. We chose this threshold based on empirical observations, though system administrators can adjust this threshold based on specific workload characteristics and service-level agreements (SLAs).

As resharding occurs, the throughput of the adaptive configurations increases. The centralized architecture reaches approximately 8,500 transactions per second, while the decentralized architecture stabilizes around 6,000 transactions per second. The No-Resharding configuration maintains a consistent throughput of about 4,000 transactions per second throughout the experiment. When the workload changes to workload B (indicated by the first vertical dashed line), both adaptive configurations experience throughput drops due to the new transaction patterns. Upon detecting these drops in throughput, they start resharding to adapt to workload B, subsequently regaining their optimal throughput levels. The No-Resharding configuration remains unaffected by the workload changes.

This pattern repeats with Workload C. The adaptive configurations adjust shard allocations in response to the new patterns, demonstrating their adaptability and regaining high-throughput levels. The centralized architecture consistently outperforms the decentralized architecture in terms of throughput and adaptation speed, attributed to its global optimization approach.

The evaluation results underscore Marlin's unique strengths. Unlike previous systems, Marlin's decentralized architecture eliminates single points of failure while achieving throughput improvements of up to 2.1x compared to non-resharded configurations. Furthermore, Marlin's dynamic sharding approach reduces communication overhead by optimizing data placement, as evidenced by a 35% improvement in throughput using hypergraph partitioning. These findings confirm that Marlin effectively bridges the gap between scalability and fault tolerance.

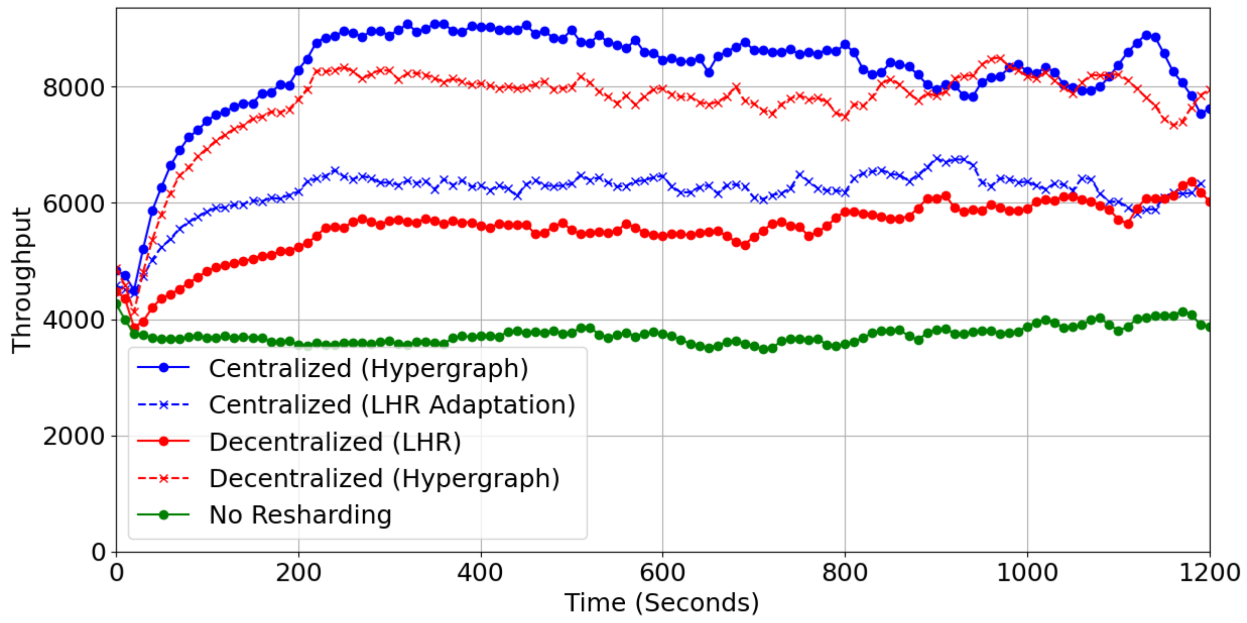

Figure 5.3: Throughputs of centralized and decentralized architectures using different partitioning schemes.

The experiment demonstrates that Marlin's adaptive resharding mechanism effectively maintains high performance under dynamic workloads. The centralized architecture adapts rapidly and achieves higher throughput due to global optimization in responding to workload changes. Despite a slower adaptation, the decentralized architecture achieves significant performance improvements without relying on a central coordinator. This resilience makes it suitable for scenarios where fault tolerance is critical and eliminating single points of failure is essential.

5.4. RQ3. Architectural Impact.

Setup. We aim to investigate the impact of the choice of architecture and partitioning strategy on the performance of Marlin.

To do so, we compare the throughput achieved by the centralized and decentralized architectures using different partitioning schemes over a single fixed workload to isolate the effects of the architecture and partitioning scheme on performance. We show these results in Figure 5.3, where each plot is a simple moving average of the throughput over time with a 5-second window.

Note that the decentralized architecture uses a Localized Heuristic Resharding (LHR) approach, which relies on local, cluster-specific observations for shard management. In contrast, the centralized architecture employs a hypergraph-based partitioning algorithm designed for global optimization. Therefore, in order to perform this experiment, we must compare them with a version of the centralized architecture that uses LHR, and a version of the decentralized architecture using the hypergraph-based partitioning approach.

The former case is relatively straightforward. For this, we extended the LHR principles to operate globally in the centralized architecture. This global adaptation applies the same resharding heuristic to the entire dataset, effectively simulating a centralized execution of the LHR strategy.

However, the latter case is more complex. Hypergraph partitioning requires global information of the entire system to optimize data placement, which is not available in the decentralized architecture. We, therefore, simulate hypergraph partitioning in the decentralized architecture. Since the workload is fixed, we first accumulate information about the data across all nodes in the system at the initialization of the experiment. We then use this information as the global state of the system to perform a one-time global optimization using the hypergraph partitioning algorithm. This lets us effectively simulate the hypergraph partitioning scheme in the decentralized architecture without requiring a central coordinator. However, we note that this cannot be done in practice if we want to maintain the decentralized nature of the architecture without sacrificing its fault tolerance.

Results. We first compare the plots where both architectures use the LHR scheme. We see that for the centralized architecture, the rate of convergence is slightly faster than that of the decentralized architecture and achieves a final throughput of slightly over 7000 transactions per second. On the other hand, the decentralized architecture takes a bit longer to converge and stabilizes at a slightly lower throughput. The gap between the performance of the two architectures is negligible and even slightly reduces over time. This indicates that for the LHR scheme, the choice of architecture has a minimal impact on performance.

In the case of the hypergraph partitioning scheme, we see that the two architectures again exhibit similar behavior. Both architectures start with similar low throughputs but converge much faster compared to the LHR scheme. There does exist a negligible gap between the performance of the two architectures, with the centralized architecture achieving slightly higher throughput. This is similar to the performances observed when using the LHR scheme. We, therefore, conclude that the choice of architecture has a minimal impact on performance given the same partitioning scheme.

Further, we observe that the throughputs achieved using the hypergraph partitioning scheme are significantly higher than those achieved using LHR. On average, the throughput is approximately 35% higher with hypergraph partitioning. This is consistent with our earlier observations in Sections 5.2 and 5.3. We thus conclude that the partitioning scheme has a more significant impact on performance compared to the choice of architecture, and attribute the differences in performance to the partitioning schemes employed by each architecture.

5.5. Ablations

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 number of clusters in the system and the percentage of cross-shard transactions in the workload.

5.5.1. Impact of Number of Clusters.

Setup. We assess Marlin's scalability by evaluating the system's throughput as we vary the number of clusters. To achieve this, we consider increasingly large numbers of clusters, from 2 to 8, 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.

Results. As shown in Figure 5.4, both the centralized and decentralized architectures scale effectively with the number of clusters. The centralized architecture consistently achieves 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 and

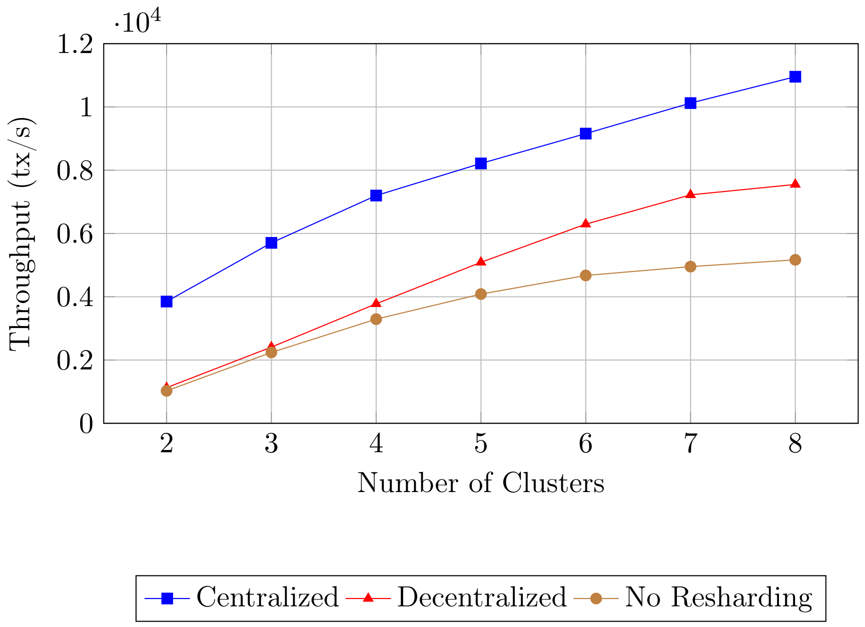

Figure 5.4: Throughput vs. Number of Clusters.

poor scalability, highlighting the importance of adaptive resharding in distributed systems.

5.5.2. Impact of Cross-Shard Transaction Rate.

Setup. To assess the effectiveness of MARLIN 's resharding mechanisms, we examine the impact of varying the percentage of cross-shard transactions on system throughput. We vary the cross-shard transaction rate in the workload over 20%, 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 with hypergraph partitioning and the decentralized architecture with the LHR scheme. The no-resharding configuration serves as a baseline for comparison.

Results. Figure 5.5 illustrates the impact of cross-shard transaction percentages on throughput across all configurations. As expected, increasing the percentage of cross-shard transactions leads to a decline in throughput due to the additional overhead required for cross-shard coordination.

The centralized architecture consistently achieves the highest throughput across all cross-shard percentages. This superior performance is attributed to its global optimization of data placement

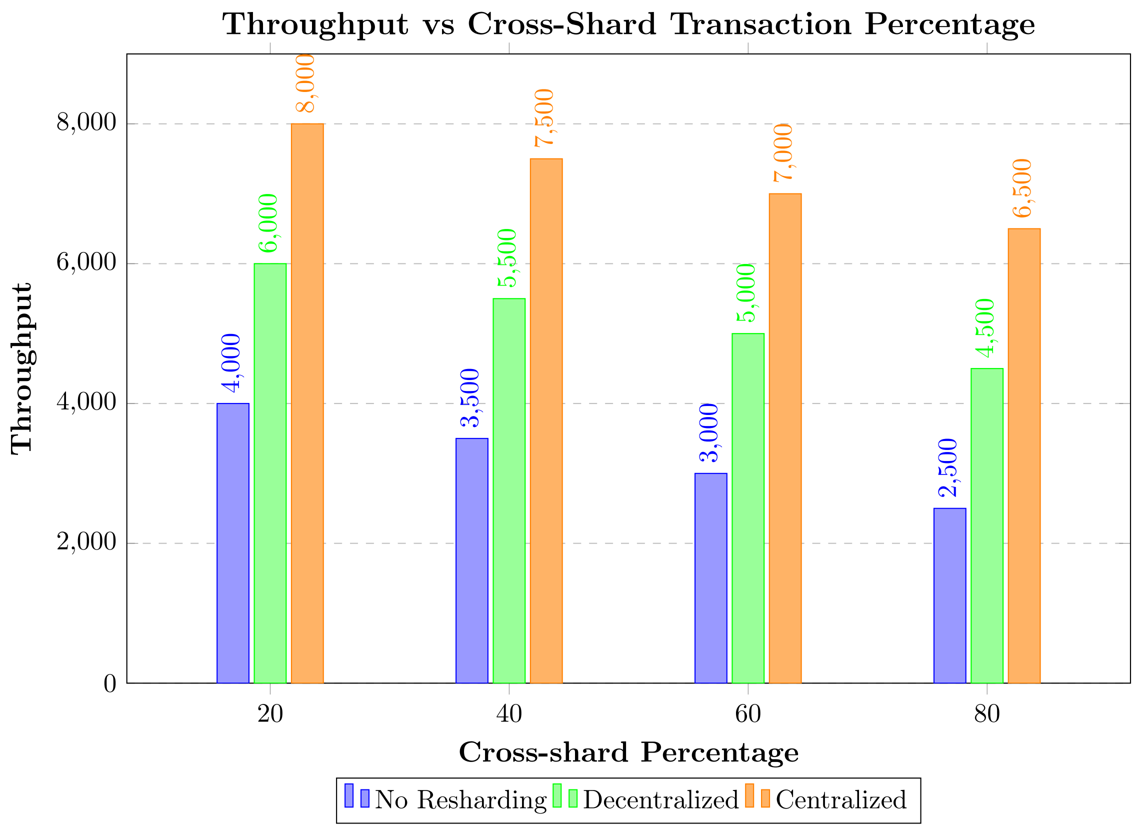

Figure 5.5: Throughput vs Cross-Shard Transaction Percentage.

using the hypergraph partitioning algorithm, which effectively minimizes cross-shard transactions by considering the entire system's transaction patterns.

The decentralized architecture also demonstrates significant improvements over the no-resharding baseline across all cross-shard percentages, particularly in moderate cross-shard scenarios (20% to 60%). This improvement highlights the effectiveness of localized heuristics in adapting to transaction patterns without relying on global information. By dynamically resharding based on local observations, the decentralized architecture mitigates cross-shard transaction overhead, achieving substantial throughput gains over the static configuration.

We also observe that at very high cross-shard transaction rates (80%), the throughput advantage of both adaptive resharding architectures decreases, as the overhead from frequent cross-shard coordination becomes a dominant factor. Nonetheless, both the centralized and decentralized architectures continue to outperform the no-resharding configuration, reaffirming the value of adaptive data placement even under heavy cross-shard workloads.

Our ablation studies demonstrate the impact of the number of clusters and cross-shard transaction rates on system performance. The experiments show that MARLIN scales effectively with the number of clusters and adapts to varying cross-shard transaction rates, achieving high throughput and efficient data placement under different workloads. The results highlight the importance of adaptive resharding in distributed systems and the benefits of global optimization in enhancing system performance.

In summary, our experimental evaluation demonstrates that MARLIN's adaptive resharding mechanism significantly improves performance in Byzantine environments for both the centralized and decentralized architectures. Our ablation studies highlight the importance of adaptive resharding and the impact of partitioning schemes on system performance. These findings demonstrate the effectiveness of MARLIN in adapting to dynamic workloads and optimizing data placement to enhance system performance in Byzantine environments.

Note: Implementation Correctness Validation During the initial experiments, throughput inconsistencies were observed, particularly when cross-shard transactions increased or decreased. These fluctuations are attributed to hardware variability and memory leaks in the experimental setup. To address these issues, the experimental framework has been entirely migrated to CloudLab, which provides a controlled and predictable environment for running distributed system experiments. A plug-and-play system has been implemented, enabling users to automatically allocate CloudLab nodes and execute experiments with a single script. This ensures reproducibility and consistency of results. Additionally, memory leaks causing crashes in Hilbit have been resolved, allowing for longer and more stable runs. UDP channel usage is now carefully managed to avoid external interference, further improving consistency. Future steps include releasing the code as open source to allow independent verification and reproducibility.

CHAPTER 6

RELATED WORK

This chapter situates Marlin within the existing literature, highlighting its unique contributions to adaptive distributed systems, Byzantine fault tolerance, and machine learning in sharding.

6.1. Adaptive Distributed Systems

Distributed systems have traditionally relied on static configurations for managing workloads. Systems like Google's Spanner Corbett et al. (2013) and Amazon's Dynamo DeCandia et al. (2007) achieve scalability through well-defined data partitioning and consistency protocols. However, these systems are not equipped to handle dynamic workload shifts or adversarial environments.

Sharding techniques, such as those employed by SWORD Quamar et al. (2013) and SharPer Amiri et al. (2021), address scalability by partitioning data across nodes. SWORD focuses on dynamic data placement but assumes a trusted environment, while SharPer introduces Byzantine fault tolerance but with static shard boundaries. Marlin extends these principles by introducing adaptive sharding mechanisms that dynamically adjust shard boundaries based on real-time workload patterns, ensuring fault tolerance in untrusted settings.

6.2. Byzantine Fault Tolerance

Byzantine fault-tolerant protocols like PBFT Castro and Liskov (1999a) and HotStuff Yin et al. (2019) enable consensus in adversarial environments but often assume static configurations. While protocols like Tendermint Kwon (2014) and Hyperledger Fabric Androulaki et al. (2018) have improved scalability, they still lack adaptability to dynamic workloads.

Marlin addresses this gap by integrating PBFT into its adaptive sharding framework, providing dynamic consensus mechanisms that maintain system reliability under adversarial conditions.

6.3. Learning-Based Optimization in Distributed Systems

Machine learning has been widely applied to optimize distributed systems. Reinforcement learning, in particular, has shown promise in dynamic system adaptation. For instance:

- NeuroShard Eldeeb et al. (2022) employs reinforcement learning for multi-objective optimization of embedding table sharding in ML systems.
- Learned database components, such as indexing Kraska et al. (2018) and scheduling Mao et al. (2018), leverage RL to adapt to changing conditions.

Marlin extends these ideas by integrating ML-driven decision-making at the system level, dynamically optimizing shard boundaries and resource allocation while maintaining Byzantine fault tolerance.

6.4. Contributions Beyond the State of the Art

Marlin bridges gaps in adaptive distributed systems and Byzantine fault tolerance by offering:

- Fully dynamic sharding mechanisms for real-time workload adaptability.
- Robust integration of PBFT for fault tolerance in untrusted environments.
- A unified approach combining adaptive mechanisms with practical, real-world scalability.

CHAPTER 7

REMAINING TASKS AND TIMELINE

This chapter outlines the remaining tasks necessary to complete this dissertation and provides a timeline for their execution. The focus is on addressing key open issues, incorporating potential VLDB reviewer feedback, ensuring the correctness and robustness of the proposed methodologies, and extending the experimental evaluation to encompass diverse workload scenarios.

7.1. Remaining Tasks

To ensure a comprehensive evaluation of the proposed framework, the following tasks remain:

7.1.1. Correctness Validation

Establishing the correctness of the proposed adaptive sharding mechanism and Byzantine fault-tolerant protocols is a primary objective. This involves:

- Conducting a **sensitivity analysis** across at least 10 parameters, such as shard size, transaction mix, failure rates, and workload skew, to validate the consistency of observed trends.
- Designing and implementing **microbenchmarks** to validate the correctness of key system components, including the two-phase commit (2PC) protocol and shard reassignment logic.
- Developing additional **unit tests** to ensure error-free operation and adherence to expected behavior under edge cases.

7.1.2. Workload Generalization

To demonstrate the robustness of the proposed framework across diverse scenarios:

- Incorporate industry-standard workloads such as **YCSB** and **TPC-C** to evaluate system performance under heterogeneous conditions.

- Extend the evaluation to multi-item transactions, quantifying the impact on convergence time and overall throughput.

- Analyze how workload shifts affect system adaptability and fault tolerance, particularly under mixed workloads involving read-heavy, write-heavy, and skewed access patterns.

7.1.3. Reproducibility and Code Release

Ensuring the reproducibility of results and promoting adoption by the research community are essential:

- Prepare a detailed **experimental setup guide**, including configurations for hardware environments (e.g., CloudLab) and workload generation.

- Provide **open-source access** to the implementation, along with documentation and pre-configured scripts to simplify replication.

- Include **logs and error-handling workflows** to highlight system resilience and ease of debugging.

7.1.4. Incorporating VLDB Reviewer Feedback

Plan to Address potential feedback we receive from VLDB reviewers:

- Allocate time to thoroughly review and address comments from the VLDB submission.

- Address any in implementing changes that enhance scalability, workload generalization, or robustness, based on the feedback.

- Revise the dissertation to reflect any changes or additions based on reviewer comments.

- Ensure any additional evaluations or adjustments align with the overall goals of the thesis.

- Resubmit the paper if needed, ensuring that all concerns have been adequately addressed and submitting the paper for publication in spring 2025.

7.1.5. Expanded Evaluation

To position this research within the broader context of distributed systems and database management:

- Perform a comparative analysis against existing adaptive sharding techniques, emphasizing scalability and fault tolerance.
- Quantify the impact of dynamic shard reassignment under Byzantine fault conditions.
- Extend results to illustrate performance on workloads with real-world characteristics, such as those derived from financial or e-commerce systems.

7.2. Timeline

The timeline below outlines the planned milestones for completing the dissertation:

- **January 2025:** Address VLDB reviewer comments, revise the dissertation, and resubmit the paper for publication.
- **February 2025:** Complete sensitivity analysis and microbenchmarking, ensuring the correctness of the proposed system.
- **March 2025:** Incorporate industry-standard workloads and evaluate system performance under diverse scenarios.
- **April 2025:** Finalize the experimental setup guide, release the codebase, and prepare for potential resubmission,
- **May 2025:** Address any concerns by committee members and defend the dissertation.

Completing these tasks will address the open challenges highlighted in this work and ensure a rigorous and comprehensive evaluation of the proposed system. The timeline emphasizes reproducibility, adaptability, and responsiveness to potential VLDB feedback, solidifying this research as a foundation for future innovations in adaptive and fault-tolerant database systems.

CHAPTER 8

CONCLUSION

This dissertation proposal addresses critical challenges in distributed systems, particularly in adaptive sharding and Byzantine fault tolerance. Through the development of Marlin, this work introduces a novel framework to enhance scalability, fault tolerance, and adaptability in untrusted and adversarial environments.

8.1. Summary of Contributions

The research presented in this proposal advances the state of the art in distributed systems by introducing Marlin, a Byzantine fault-tolerant system with adaptive sharding capabilities. The key contributions include:

- **Adaptive Sharding Framework:** Marlin dynamically adjusts shard boundaries based on real-time workload characteristics, minimizing cross-shard communication and optimizing performance.

- **Fault Tolerance with PBFT:** The integration of PBFT for intra- and cross-shard consensus ensures robust fault tolerance, even in environments with malicious nodes.

- **Comprehensive Evaluation:** Rigorous evaluations demonstrate Marlin's adaptability, scalability, and fault tolerance across diverse workloads and adversarial conditions.

8.2. Broader Reflections and Implications

This research holds broader implications for the design and deployment of distributed systems:

- **Scalability in Untrusted Environments:** Adaptive mechanisms in Marlin demonstrate the ability to maintain high performance and fault tolerance in Byzantine environments.

- **Dynamic Workload Adaptability:** Marlin's adaptability to shifting workload patterns provides insights for real-world applications such as decentralized finance, blockchain systems, and e-commerce platforms.

- **Trade-offs in Decentralized Architectures:** The evaluation of centralized and decentralized architectures informs future system designs that balance performance, resilience, and complexity.

8.3. Future Research

Building on Marlin's success, future research could explore:

- Integration with emerging consensus protocols like HotStuff for further reducing communication overhead.

- Enhanced heuristic models for faster shard reassignment in latency-sensitive environments.

- Extending Marlin's framework to support heterogeneous workloads, ensuring broader applicability in diverse domains such as blockchain and IoT.

8.4. Conclusion

This dissertation represents a step forward in addressing the challenges of scalability, adaptability, and fault tolerance in distributed systems. By combining adaptive mechanisms with robust fault-tolerant protocols, Marlin establishes a foundation for designing next-generation distributed systems that can thrive in dynamic and adversarial environments. The methodologies and findings presented in this work provide a roadmap for advancing the field of distributed systems and offer practical solutions for real-world applications.

APPENDIX A

END-TO-END EXAMPLE

To illustrate the resharding process, consider a system with three shards and the following transactions:

Transaction ID	**Accessed Keys**
T1	{A, B}
T2	{C, D}
T3	{C, D}
T4	{E}
T5	{A, B}
T6	{B, A}
T7	{E, F}
T8	{E, G}

Table A.1: Transactions and their accessed keys

Initial Key Distribution With this distribution, many transactions are cross-shard, resulting in high communication costs.

Shard	S1	S2	S3
Keys	A, E, F	B, C	D, G

Table A.2: Initial key distribution across shards

Proposal Generation Steps In the decentralized setup each cluster generates proposals to transfer keys based on its previous transaction history. Following is an example for Shard S1 -

(a) *Transaction Frequency Sorting*: Based on keys, each node in S1 cluster records transactions {T1, T4, T5, T6, T7, T8} since they have one or more keys from {A, E, F}. T4 gets filtered since it only involves one key. Remaining transactions are sorted based on frequency of accessed key combinations-

Accessed Keys	**Frequency**	**Shards Involved**
{A,B}	3	S1, S2
{E,F}	1	S1
{E,G}	1	S1, S3

Table A.3: Transactions sorted by frequency

(b) *Data Assignment*: S1 cluster's leader proposes moving key {A} to S2 to co-locate frequently accessed pairs. It skips moving {E}, since it parses transaction {E, F} first and decides to keep both keys in S1.

Shard	S1	S2	S3
Keys	E, F	A, B, C	D, G

Table A.4: Key distribution after co-locating A and B

(c) *Proposal Submission*: S1 cluster's leader broadcasts the resharding proposal to other nodes in its cluster.

Proposal Validation at Replica Nodes In this step each S1's non-leader nodes validate resharding proposal from S1's cluster leader. This ensures we handle the case with a Byzantine fault in leader.

(a) *Fetch transactions* {T1, T4, T5, T6, T7, T8}

(b) *Compute new cross-shard distribution* Since A and B are colocated the only cross-shard transaction remaining is T8. Thus the number of cross-shard transactions reduces from 4 to 1.

(c) *Sign proposal* Each replica node signs the proposal to attest its been validated.

Proposal Evaluation at Receiving Shards Each receiving shard processes the proposal as follows:

(a) *Simulation*: Each shard simulates applying the proposed data movements to its local Data Shard Map.

(b) *Batch Evaluation*: Shards evaluate the impact of moving data items on cross-shard transaction rates and load balance.

(c) *Acceptance Criteria*: If a shard verifies that the proposed movements reduce cross-shard transactions without violating shard size constraints ($S_{\min}$ and $S_{\max}$), it accepts the data movements and signs the proposal.

For example, **Shard 2** evaluates moving key A to its shard. It loads the transactions it received from its snapshot {T1, T2, T3, T5, T6}:

- Before Proposal: Shard 2 manages keys B, C.
- Proposal: Move key A to Shard 2.
- After Simulation: Shard 2 manages A, B, C with size within limits.
- Result: Reduces number of cross-shard transactions from 5 to 2 without overloading Shard 2.
- Action: Shard 2 accepts the proposal and signs it.

Final Shard Allocation After receiving signatures from the required number of nodes, the initiating node finalizes and executes the resharding plan. This new distribution reduces global cross-shard transactions from 6 to 3, optimizing performance while maintaining balanced shard sizes. The number of cross-shard transactions will further reduce as other shards also repeat the resharding process.

BIBLIOGRAPHY

Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. SharPer: Sharding permissioned blockchains over network clusters. In *SIGMOD Int. Conf. on Management of Data*, pages 76–88. ACM, 2021.

Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. Qanaat: A scalable multi-enterprise permissioned blockchain system with confidentiality guarantees. *Proc. of the VLDB Endowment*, 15(11):2839–2852, 2022.

Mohammad Javad Amiri, Ziliang Lai, Liana Patel, Boon Thau Loo, Eric Lo, and Wenchao Zhou. Saguaro: An edge computing-enabled hierarchical permissioned blockchain. In *Int. Conf. on Data Engineering (ICDE)*, pages 259–272. IEEE, 2023.

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

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

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

Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *Computing Surveys (CSUR)*, 13(2):185–221, 1981.

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

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, and Harry Li. Tao: Facebook's distributed data store for the social graph. In *Annual Technical Conf. (ATC)*, pages 49–60. USENIX Association, 2013.

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

Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX Association, 1999a.

Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, USA, February 1999b. USENIX Association. ISBN 978-1-880446-39-3.

Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 9, pages 153–168. USENIX Association, 2009.

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

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

Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *SIGMOD Int. Conf. on Management of Data*, pages 123–140. ACM, 2019.

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

Tamer Eldeeb, Zhengneng Chen, Asaf Cidon, and Junfeng Yang. Neuroshard: towards automatic multi-objective sharding with deep reinforcement learning. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–12, 2022.

Ethan Frey and Christopher Goes. Cosmos inter-blockchain communication (ibc) protocol. https://cosmos.network. 2018.

Daniel Golovin, Gabor Bartok, Eric Chen, Emily Donahue, Tzu-Kuo Huang, Efi Kokiopoulou, Ruoyan Qin, Nikhil Sarda, Justin Sybrandt, and Vincent Tjeng. Smartchoices: Augmenting software with learned implementations. *arXiv preprint arXiv:2304.13033*, 2023.

R. Guerraoui, N. Gupta, and R. Pinot. Byzantine machine learning: A primer. *Communications of the ACM*, 2024. doi: 10.1145/3616537.

N. Gupta, S. Liu, and N.H. Vaidya. Byzantine fault-tolerant distributed machine learning using stochastic gradient descent (sgd). *ACM Transactions on Parallel Computing*, 7:1–15, 2020. URL https://arxiv.org/abs/2008.04699.

Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Learning a partitioning advisor for cloud databases. In *SIGMOD Int. Conf. on Management of Data*, pages 143–157. ACM, 2020.

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

Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196909. URL http://doi.acm.org/10.1145/3183713.3196909.

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

Jae Kwon. Tendermint: Consensus without mining. 2014.

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

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

Yimeng Liu, Yizhi Wang, and Yi Jin. Research on the improvement of mongodb auto-sharding in cloud environment. In *Int. Conf. on Computer Science and Education (ICCSE)*, pages 851–854. IEEE, 2012.

Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. *arXiv:1810.01963 [cs, stat]*, 2018. URL http://arxiv.org/abs/1810.01963. arXiv: 1810.01963.

Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning. In *Proceedings of the 37th ACM Special Interest Group in Data Management*, SIGMOD '18, Houston, TX, 2018. doi: https://doi.org/10.1145/3183713.3196935.

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

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

Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. Fast and effective distribution-key recommendation for amazon redshift. *Proc. of the VLDB*

Endowment, 13(12):2411–2423, 2020.

Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, et al. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Symposium on Operating Systems Principles (SOSP)*, pages 18–34. ACM, 2021.

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

Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proc. of the European Conf. on Computer Systems (EuroSys*, pages 1–16, 2020.

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

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

Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *PVLDB*, 8(3):245–256, 2014a. ISSN 2150-8097. doi: 10.14778/2735508.2735514. URL http://dx.doi.org/10.14778/2735508.2735514.

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

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

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

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

Xuanhe Zhou, Guoliang Li, Jianhua Feng, Luyang Liu, and Wei Guo. Grep: A graph learning based database partitioning system. *Proc. of the ACM on Management of Data*, 1(1):1–24, 2023.