# Marlin: Efficient Coordination for Autoscaling Cloud DBMS

WENJIE HU, University of Wisconsin-Madison, USA
GUANZHOU HU, University of Wisconsin-Madison, USA
MAHESH BALAKRISHNAN*, Meta, Inc., USA
XIANGYAO YU, University of Wisconsin-Madison, USA

Modern cloud databases are shifting from converged architectures to storage disaggregation, enabling independent scaling and billing of compute and storage. However, cloud databases still rely on external, converged coordination services (e.g., ZooKeeper) for their control planes. These services are effectively lightweight databases optimized for low-volume metadata. As the control plane scales in the cloud, this approach faces similar limitations as converged databases did before storage disaggregation: scalability bottlenecks, low cost efficiency, and increased operational burden.

We propose to disaggregate the cluster coordination to achieve the same benefits that storage disaggregation brought to modern cloud DBMSs. We present Marlin, a cloud-native coordination mechanism that fully embraces storage disaggregation. Marlin eliminates the need for external coordination services by consolidating coordination functionality into the existing cloud-native database it manages. To achieve failover without an external coordination service, Marlin allows cross-node modifications on coordination states. To ensure data consistency, Marlin employs transactions to manage both coordination and application states and introduces MarlinCommit, an optimized commit protocol that ensures strong transactional guarantees even under cross-node modifications. Our evaluations demonstrate that Marlin improves cost efficiency by up to 4.4× and reduces reconfiguration duration by up to 4.9× compared to converged coordination solutions.

CCS Concepts: • **Information systems → Relational parallel and distributed DBMSs**; **Distributed database transactions**; **Autonomous database administration**.

Additional Key Words and Phrases: Cloud DBMS coordination, Autoscaling, Storage disaggregation

**ACM Reference Format:**
Wenjie Hu, Guanzhou Hu, Mahesh Balakrishnan, and Xiangyao Yu. 2025. Marlin: Efficient Coordination for Autoscaling Cloud DBMS. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 255 (September 2025), 28 pages. https://doi.org/10.1145/3749173

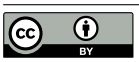
## 1 Introduction

Modern cloud databases are moving from monolithic, converged architectures—where compute and storage reside on the same physical servers [31, 46, 88, 96]—to storage disaggregation, where compute and storage are separated and connected via network [7, 15, 27, 29, 45, 91, 101]. This disaggregation enables many benefits over the converged design, such as independent and elastic scaling of compute and storage, improved resource utilization, reduced cost, and flexible deployment.

Despite modernizing their data planes through storage disaggregation, cloud databases still rely on external, centralized, and converged coordination services (e.g., ZooKeeper [51], etcd [1],

*Work done while unaffiliated.

Authors' Contact Information: Wenjie Hu, University of Wisconsin-Madison, Madison, Wisconsin, USA, wenjiehuhippo@cs.wisc.edu; Guanzhou Hu, University of Wisconsin-Madison, Madison, Wisconsin, USA, guanzhou.hu@wisc.edu; Mahesh Balakrishnan, Meta, Inc., Menlo Park, California, USA, mbalakrishnan@meta.com; Xiangyao Yu, University of Wisconsin-Madison, Madison, Wisconsin, USA, yxy@cs.wisc.edu.




ACM 2836-6573/2025/9-ART255
https://doi.org/10.1145/3749173

Chubby [16]) to maintain shared state for their control planes. In effect, these services are miniature databases optimized for storing small-scale coordination states. However, as control plane workloads scale in size and throughput in tandem with user data, these coordination services face similar limitations as converged databases did before storage disaggregation. In particular, conventional coordination suffers from three major limitations: ❶ **Scalability bottleneck.** Converged coordination services for low-scale metadata typically adopt single-master design [1, 16, 51] where all traffic is routed to a single node. The problem becomes more pronounced on the cloud as frequent reconfigurations (e.g., rescaling, rebalancing) amplify coordination workloads. Moreover, cloud databases often employ geo-distributed deployments [83]. The cross-region latency further increases the performance overhead of a centralized coordination service. ❷ **Low cost-efficiency.** Cloud workloads change rapidly and are often bursty, with peak system utilization many times higher than the 99th percentile [30, 44, 92]. Static coordination services are commonly overprovisioned to handle peak coordination demand [26], resulting in underutilized resources and poor cost efficiency. Furthermore, the converged design prevents true "scale-to-zero" elasticity when workloads diminish (e.g., ZooKeeper requires at least three VMs for reliability), incurring significant upfront costs. ❸ **Operational burden.** Relying on an external, converged coordination service complicates deployment and maintenance. Operators must provision and manage this additional subsystem before deploying the database. Unlike stateless compute, converged coordination services must remain stateful, fault-tolerant, and always on, requiring specialized expertise to configure and operate. Moreover, external dependencies can reduce overall availability and increase troubleshooting efforts. According to the "golden rule" of component reliability [85], any critical dependency must be 10 times as reliable as the overall system's target to ensure its contribution to overall unreliability is negligible. Anecdotally, coordination services are notoriously difficult to operate reliably [10, 20, 73] because they implement complex consensus protocols directly.

Prior works partially address some of the limitations but often exacerbate others, and therefore cannot fundamentally resolve the problem. One approach leverages an external distributed transactional database system (DDBMS) as the coordination service to improve scalability via partitioning [17, 33]. However, it significantly complicates the architecture, inflating the cost and introducing greater external dependencies and heavier operational burdens. Moreover, it can increase latency as each coordination request requires more network hops to traverse DDBMS's internal hierarchy. Another approach [26] reduces cost by running a coordination service on serverless functions. Nevertheless, it does not address the scalability bottleneck, and serverless functions are inherently unsuitable for performance-critical tasks. Other works try to solve all three limitations by utilizing internal infrastructure to eliminate external coordination services [23, 24, 96]. However, they focus on converged architectures where compute nodes can be protected by the storage layer's replication protocol. They are therefore inapplicable to cloud-native databases with compute-storage disaggregation, where the replication protocol resides solely in the storage layer and is not accessible at the compute layer [47, 76].

In this paper, we aim to answer the following key question: *Can we disaggregate the cluster coordination to achieve the same benefits that storage disaggregation brought to the data planes of modern cloud DBMSs?* Our answer is Marlin, a cloud-native coordination mechanism that fully embraces storage disaggregation. Marlin eliminates the need for external coordination services by consolidating coordination functionality into the database it manages: coordination states are stored in the database's highly available, disaggregated storage layer, while consistent coordination is handled in the compute layer using standard transaction managers. This design enables linear scalability through partitioned access paths, reduces costs via independent scaling of compute and storage, and simplifies operations by removing external dependencies. The design of Marlin faces the following key challenges:

**Challenge #1: Scalability.** An intuitive approach for scalable coordination is to partition coordination states across compute nodes. However, without an external coordination service, this exclusive ownership can cause a "failover deadlock": if a node owning failover-critical metadata fails, the remaining nodes cannot update that metadata to promote a new owner, stalling the failover process. Marlin solves this by allowing multiple nodes to access failover-critical states, which sit in the disaggregated storage. Marlin identifies two types of coordination states essential to failover: (1) *Data ownership*, mapping data partitions to their owner nodes, and (2) *Group membership*, listing active nodes in the cluster. Data ownership information, which grows with user data, is partitioned by the owner node ID. Each node manages its own partition under normal conditions, while permitting other nodes to modify it if the owner fails. Group membership, which is relatively small and updated infrequently, is stored in a single, non-partitioned log accessible by all nodes. Other states unrelated to failover (e.g., security keys, monitoring metrics) are partitioned like regular user data and exclusively managed by their respective owner nodes at all times.

**Challenge #2: Data Consistency.** Marlin must ensure data consistency without relying on an external coordination service, even during concurrent reconfigurations or arbitrary compute node failures. Marlin reuses existing database components (e.g., transaction and cache managers) and manages all coordination state accesses through special *reconfiguration transactions*.

Conventional transaction protocols (e.g., 2PL, OCC, and 2PC) cannot fully resolve metadata inconsistencies because their assumption that a compute node has exclusive control over the data it manages no longer holds in Marlin. For example, group membership metadata may be modified by multiple compute nodes concurrently, and compute node failures or network partitions can create ambiguous data ownership. To solve this, Marlin introduces MarlinCommit, an optimized commit protocol that ensures strong transactional guarantees even when multiple nodes can access the same data. The key insight is to leverage a conditional append API, *Append(updates, LSN)*, which succeeds only if the current log version matches the expected version. Via this API, MarlinCommit can detect cross-node modifications and commit a transaction only if no cross-node modifications have been made. This approach is storage-agnostic and can be supported by any disaggregated storage services offering compare-and-swap functionality [21, 29, 40, 65, 68, 79].

In summary, the paper makes the following key contributions:

- We revisit coordination solutions in modern cloud OLTP databases and identify their limitations in the context of autoscaling.
- We propose Marlin, a cloud-native coordination mechanism for modern cloud OLTP databases that is self-contained, scalable, and cost-efficient. Marlin eliminates the need for an external coordination service by integrating the coordination into the database itself.
- We implement Marlin in a cloud OLTP DBMS testbed. Our evaluations show that Marlin improves cost-efficiency by 4.4× and reduces reconfiguration duration by 4.9× compared to a converged coordination service.

The paper is organized as follows: §2 discusses architectural trends in cloud databases and provides background on cluster coordination. §3 outlines the system model and §4 details Marlin's core design, including coordination state management, primitives, and their usage examples. §5 describes implementation details of the database testbed and Marlin. §6 evaluates Marlin's performance, §7 discusses related work, and §8 concludes the paper.

## 2 Background and Motivation

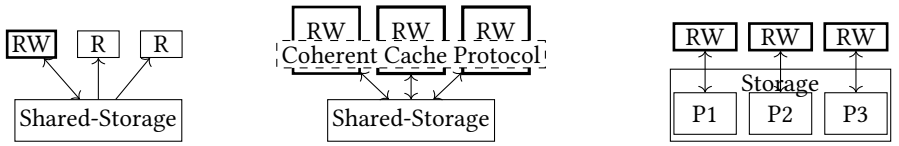
This section first describes the architecture of storage disaggregation (§2.1). Then, it explores the taxonomy of scalable cloud OLTP archetypes based on data access paths (§2.2). Finally, it discusses the limitations of cluster coordination solutions in existing cloud DBMSs (§2.3).

### 2.1 Storage Disaggregation in Cloud DBMS

Modern cloud-native DBMSs [7, 27, 29, 45, 69, 91] increasingly adopt a storage disaggregation architecture, where storage and compute services operate independently, connected via the data center network. This design contrasts with conventional shared-nothing systems that rely on direct-attached storage. Storage disaggregation enables independent scaling of compute and storage resources, enhancing elasticity and cost efficiency. The disaggregation model also supports limited computation within the storage layer, which can be performed on the storage nodes [91] or in a layer close to the storage nodes [4, 6]. Prior research has leveraged near-storage computation to enhance performance and reduce costs [47, 95]. Building on this concept, Marlin employs near-storage computation capabilities (i.e., Compare-And-Swap) for coordination purposes.

### 2.2 Cloud OLTP DBMS

According to the taxonomy recently proposed by Tobias et al. [102], contemporary scalable cloud OLTP DBMSs fall into three archetypes, categorized by the distinct data access paths in data planes, as shown in Figure 1.




(a) Single-Writer (b) Shared-Writer (c) Partitioned-Writer

Fig. 1. Three Archetypes of Cloud OLTP Databases.

**Single-Writer** archetype employs a single read-write node (RW-node) with multiple read-only nodes on top of the shared storage and is prevalent in commercial cloud databases [18, 18, 45, 63, 72, 89, 91]. While being simple to manage, this archetype suffers from write-heavy workloads, as the single RW-node prohibits asymptotic scalability [102]—a concept that measures how well a system scales w.r.t the number of nodes for basic data access operations when increasing the load in the system. **Shared-Writer** archetype allows multiple RW-nodes to modify the same set of data items on the shared-storage, hence improving write scalability. Because data must be kept consistent across multiple nodes, these systems need to address the buffer-cache coherence problem [70, 78]. However, due to the complexity of maintaining consistency and cache coherence, this approach is limited to a few commercial shared-disk systems [28, 75] and is primarily explored in research prototypes [14, 100, 103]. **Partitioned-Writer** archetype partitions the database across multiple RW-nodes, each responsible for writes on an exclusive subset of data. This approach is widely used in modern commercial DBMSs [23, 27, 29, 40, 50, 99, 101] as it offers asymptotic scalability of both reads and writes and reduces the complexity of Shared-Writer architectures. These databases typically use the two-phase commit protocol to ensure the atomicity of distributed transactions. Partitioned-Writer can be achieved either on local storage [23, 96] or disaggregated storage [29, 101], with the latter being more prevalent in cloud-native databases. The storage choice drives two key differences: ❶ Disaggregated storage allows independent scaling and billing of compute and storage resources, while local storage tightly binds compute and storage to the same node. ❷ With disaggregated storage, RW-nodes need no replication because they are stateless and the storage layer is highly available. In contrast, local storage systems must replicate nodes for fault tolerance since compute and storage are coupled. We use Partitioned-Writer archetype with storage disaggregation as the target cloud OLTP DBMS for Marlin because it fully achieves asymptotic scalability and is widely used in commercial cloud databases.

### 2.3 Cluster Coordination in Cloud DBMS

We categorize coordination solutions in existing cloud DBMSs into a quadrant chart in Figure 2 by two dimensions: ① whether the coordination is integrated into the base system, and ② whether the coordination service embraces storage disaggregation.

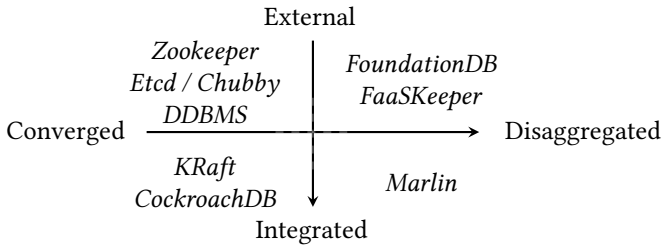
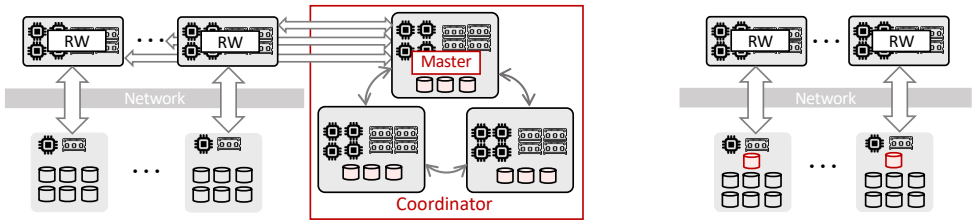



Figure 2. Solution Quadrants for Cluster Coordination.

As shown in the top left quadrant, although many cloud databases implement their data planes following the storage disaggregation architecture, many of these databases [21, 22, 27, 50, 54, 56, 71, 94] continue to rely on **external** and **converged** coordination services such as ZooKeeper [51], Etcd [1], or Chubby [16] for their control planes. However, in cloud environments, as cluster sizes and data volumes grow, metadata size expands rapidly[1]. This rising demand for elasticity necessitates frequent rescaling, rebalancing, and dynamic repartitioning, further increasing the number of accesses to coordination states. For instance, a database with 1 PB of user data with 32 MB partitions would require approximately 32 GB metadata to map partitions to owner nodes, assuming each map entry consumes 1 KB. The size of metadata grows even more with the adoption of advanced data partitioning strategies, such as fine-grained or dynamic partitioning [32, 58, 60, 77, 82, 87].




(a) Cloud DBMS with An External and Converged Coordination Service (b) Cloud DBMS with Marlin

Fig. 3. **Cluster Coordination in Partitioned-Writer DBMSs.** — Using external, centralized, and converged coordination services presents limitations: (1) scalability bottleneck (2) low cost-efficiency, and (3) increased operational burdens.

As exemplified in Figure 3a, existing external and converged coordination services typically take a centralized (single-master) architecture. This design can cause scalability issues as all coordination requests must pass through the master node. Moreover, maintaining an external coordination service for bursty coordination requests is not cost-efficient and adds operational burdens.

Previous works have explored ways to alleviate scalability and cost issues in external coordination solutions. To improve scalability, Snowflake [33] and ByConity [17] replace the single-master coordinator with FoundationDB [101], thereby achieving scalability through data partitioning. To reduce cost, systems such as FaaSKeeper [26] run coordination services on serverless functions and

[1]In this paper, "coordination states" and "metadata" are used interchangeably.

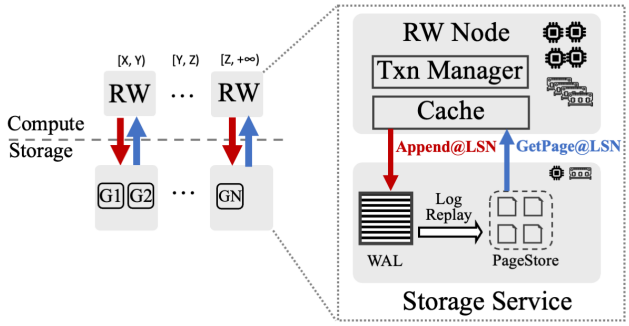



Fig. 4. Reference Cloud Database. It follows key design principles [76, 102] of modern scalable cloud-native databases: ❶ Storage disaggregation, ❷ Partitioned-Writer archetype, and ❸ Log-as-the-database paradigm.

disaggregate the compute and storage resources. However, these solutions still rely on external coordination services and fail to fully resolve the three key limitations outlined in §1. Specifically, FoundationDB cannot automatically scale up and down as coordination loads fluctuate. It also incurs additional costs for maintaining an external coordination service and introduces external dependencies and operational burdens. FaaSKeeper does not address scalability and suffers from limitations inherent to serverless platforms, such as cold-start latency, unpredictable performance, and the lack of persistent storage, which are unsuitable for coordination tasks.

As shown in the bottom left quadrant of Figure 2, recent works like CockroachDB and KRaft [23, 24, 96] try to solve the three limitations by **integrating** the coordination into the database's internal infrastructure. For instance, CockroachDB [23] partitions metadata into ranges and employs a two-level structure for efficient key lookups. The first level stores metadata ranges and acts as the "metadata for metadata" and the second level addresses user data. Each metadata range is managed by a Raft group. The fundamental difference between our work and CockroachDB-like solutions is the *architectural assumption* of the database system: whether it employs **storage disaggregation**. Systems like CockroachDB assume a converged architecture where compute and storage are coupled on the same node, which is replicated through consensus protocols. In a storage disaggregation architecture, however, only the storage layer has built-in replication but the compute nodes are stateless and not replicated. Therefore, when a compute node fails, the coordination mechanism must carefully handle the coordination states through interactions between the remaining compute nodes and the storage layer—a challenge that does not exist in converged systems.

In this paper, we aim to address prior limitations of conventional coordination services by developing a cloud-native coordination mechanism suitable for storage-disaggregated databases.

## 3 System Model

Our target database architecture, illustrated in Figure 4, adopts the Partitioned-Writer archetype with storage disaggregation. The storage layer provides reliable data storage with limited near-storage compute capability, and the compute layer performs transaction processing and data access. Application data is partitioned across the cluster, with each partition assigned to a single compute node. For clarity, this section focuses on the scalable, cloud-native OLTP architecture without discussing the coordination mechanism.

### 3.1 Storage Layer

The disaggregated storage layer provides highly available and fault-tolerant storage for the persistent image of the database. As shown in Figure 4 (right), the system follows the log-as-the-database

(LogDB) paradigm [76], a design principle prevalent in storage-disaggregated databases [7, 36, 45, 91]. The compute nodes append updates to write-ahead logs (WAL) on the storage layer upon transaction commit, establishing these logs as the ground truth of the database. The storage materializes WAL into the data pages asynchronously through the log replay service, eliminating the need to write back dirty pages from compute nodes. In prevalent LogDBs [7, 36, 76, 91], the storage service provides two standard APIs: (1) *Append(updates)* to append updates to the tail of WAL, and (2) *GetPage(pageId, LSN)*, denoted *GetPage@LSN* [7], to retrieve a page from page stores that has applied all updates up to the expected log sequential number (LSN). Leveraging the near-storage computation capabilities (i.e., Compare-And-Swap), Marlin enhances the standard append API into *Append(updates, LSN)*, a conditional append operation that can succeed only if the log end has that particular LSN at the time of append. We mark this operation as *Append@LSN* and use it to enable coordination, which will be further discussed in §4.3.

### 3.2 Compute Layer

Each compute node can serve both reads and writes. The compute nodes are stateless and are not replicated. On the write path, an update is sent to the WAL using *Append@LSN* upon transaction commit and then updates its local cache. If the cache needs to evict a dirty page, the node simply deletes it without writing it back to the storage layer. The compute node also tracks the highest LSN committed to the WAL, denoted *H-LSN*. On the read path, the compute node first attempts to retrieve the page from its local cache. If there is a cache miss, it fetches the missing page from the page store using *GetPage@LSN* for the newest version *H-LSN*.

## 4 Marlin

In contrast to relying on an external, converged coordination service for orchestrating the cloud DBMS (Figure 3a), Marlin integrates coordination functionalities into the database itself to remove the external dependency. This section presents the core design of Marlin. §4.1 discusses some candidate design options to achieve this integration idea and proposes Marlin's solution to store and maintain coordination states. §4.2 and §4.3 explain how Marlin accesses and modifies these states to ensure consistent coordination. §4.4 provides representative examples, including scale-out and node failover, to illustrate how Marlin achieves cluster management for cloud OTLP databases. §4.5 summarizes the correctness invariants.

### 4.1 Coordination State Management

One way to integrate coordination into the cloud database is to store coordination states in a single-partition user table, managed exclusively by one compute node. While easy to implement, the owner node of the coordination states becomes a scalability bottleneck. Moreover, this exclusive access model introduces a "failover deadlock": if the metadata owner node fails, other nodes cannot safely update the critical metadata to promote a new owner. This stalls the failover process and leads to system-wide unavailability. To ensure safe failover, either an external coordination service or a specialized coordination protocol is required.

An alternative design is to manage coordination states as a partitioned user table: states are partitioned and stored in the disaggregated storage layer, with each partition managed exclusively by an owner compute node. This design requires a two-level metadata structure, where a top-level metadata partition is used to locate lower-level metadata partitions. While mitigating the scalability issue, it still suffers from the same failover problem.

To address this failover issue without relying on external coordination services, Marlin manages critical coordination states using special system tables that allow multiple compute nodes to read and update when necessary. Specifically, Marlin identifies and organizes two types of coordination

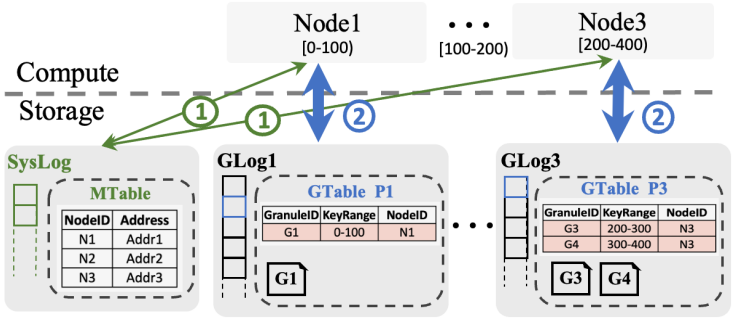



Fig. 5. System Tables (MTable and GTable).

states into special system tables: (1) *Group membership*, which lists the operational compute nodes in the cluster, is stored in the *membership table* (MTable), and (2) *Data ownership*, the mapping between data partitions and their owner compute nodes, is stored in the *granule ownership table* (GTable). Granules are essentially fine-grained partitions with a small size (64 KB in the current implementation). Figure 5 illustrates schemas of these system tables. Each row in MTable contains a node ID and the corresponding server address, while each row in GTable specifies a granule ID, its key range, and the ID of the compute node that owns it. Because these states are critical for coordinating node failover, they must remain available for both read and write operations under arbitrary compute node failures.

MTable is typically small in size and remains unpartitioned. All modifications to it are recorded in a single log, SysLog, as illustrated in Figure 5. To prevent the failover problem, SysLog has no exclusive owner, allowing all compute nodes to access and modify it (①). In contrast, GTable grows linearly with application data volume and may experience bursty access during large-scale cluster reconfigurations. Therefore, Marlin partitions GTable by the granule owner node ID to improve scalability. Each GTable partition is managed by the same node as the application data granules it describes, resulting in a **flat** metadata design and obviating the need for a two-level "metadata-for-metadata" structure. GTable keeps that locality property even after reconfigurations since it is automatically and spontaneously repartitioned through migrations. Under normal circumstances, each GTable partition is accessed exclusively by its owner node (②), and modifications are recorded in a per-node WAL (GLog). If the owner node fails, Marlin allows other compute nodes to access the partition, allowing the failover to progress smoothly. Other coordination states unrelated to node failover—such as security keys, ACLs, permissions, monitoring metrics, and user profiles—are managed by Marlin similarly to application data. Marlin organizes these states as regular partitioned tables, with each partition exclusively accessed by its owner compute node at all times.

### 4.2 Coordination Via Transactions

Marlin must ensure data consistency under autoscaling context where system reconfigurations (e.g., scaling and load balancing), user transactions, and compute node failures can occur concurrently. Each of these activities involves multi-step operations that access or modify application and coordination states and should be performed together as a single logical unit. Therefore, Marlin manages all state access, including coordination and application states, through transactions. ACID properties are guaranteed for all transactions, which will be discussed in §4.3. This section focuses on how to compose transactions on system tables and user tables to support consistent coordination.

Marlin provides two types of transactions. *User transactions* are regular user-defined transactions. They can run interactively or as stored procedures. *Reconfiguration transactions* handle operations on coordination states (i.e., system tables) and are predefined stored procedures that take parameters.

| Transaction | Description |
|---|---|
| **AddNodeTxn** | Add a new node into the cluster |
| **DeleteNodeTxn** | Remove an existing node from the cluster |
| **MigrationTxn** | Migrate one or a list of granules from a source node to a destination node |
| **RecoveryMigrTxn** | Migrate one or a list of granules from an unresponsive node to a destination node |
| **ScanGTableTxn** | Query the data ownership of the full cluster |

Table 1. Reconfiguration Transactions.

As shown in Table 1, five types of reconfiguration transactions serve as the core building blocks for common autoscaling scenarios, including scale-out/in, load balancing, and failover. The first two modify MTable, while the remaining three operate on GTable. All transactions in Marlin are detailed in Algorithm 1. RPC requests to peer nodes are expressed using the notation: $result \leftarrow RPC^{node}_{sync/async}::Request()$, where the superscript *node* indicates the target node and the subscript *sync*/*async* denotes whether the RPC is synchronous or asynchronous.

The first four reconfiguration transactions are designed to modify system tables. *AddNodeTxn* is executed on the new node to be added and committed to SysLog. *DeleteNodeTxn* is executed on the node to be deleted or on the node detecting a failure. *MigrationTxn* is a cross-node transaction that is executed on both the source (*src*) and destination (*dst*) nodes involved in a granule migration. *RecoveryMigrTxn* is a single-node transaction that is executed only on *dst* node for the migration. However, it commits on GLog of both *src* and *dst* nodes. It is the key component to cope with failover. These reconfiguration transactions are composed of three main steps:

(1) **Check the data effectiveness** (lines 8, 14, 20-21, 28-29): Verify the system tables to ensure the database is in a valid state for the current reconfiguration. This step prevents data corruption during concurrent reconfigurations. For instance, *DeleteNodeTxn* validates that the target node remains a member of the cluster before proceeding with deletion. Note that user transactions also incorporate this verification to ensure that data ownership is not altered by concurrent reconfiguration transactions. Before accessing application data, each *UserTxnRequest* confirms that the granule to be accessed is owned by the current node by checking the GTable (lines 2-3).
(2) **Modify configuration states** (lines 11, 15, 22-23, 30): Update the corresponding system tables to reflect the desired changes.
(3) **Commit the transaction** (lines 12, 16, 24, 31): Commit the updates to the logs associated with the system tables to ensure durability. Marlin utilizes MarlinCommit to ensure atomic commit, which will be discussed shortly.

The last reconfiguraion transaction is a read-only *ScanGTableTxn*, which serves routing purposes. It is triggered by routers to locate partition owners. It is a distributed transaction executed across all compute nodes to perform a full scan of the GTable. It will not cause much performance overhead as it does not need to be invoked per request, and routers can cache the scan results. Cache staleness in routers does not compromise system correctness, as Marlin ensures each compute node maintains the ground truth for its owned GTable partition. Consequently, if a request is misrouted due to stale routing information, the receiving node can detect that it no longer owns the granule (lines

**Algorithm 1: User and Reconfiguration Transactions** — Difference between Marlin *UserTxn* and standard *UserTxn* is highlighted in gray.

```
1  Function UserTxnRequest(key)
       # Acquire a read lock on GTable for the granule if using 2PL
2      nid ← GTable[getGranuleID(key)].NodeID
3      if cur_node is nid then
4          executeUserRequest()
5      else
6          Abort and return WrongNodeError(nid)

7  Function AddNodeTxn(new_node)
8      if MTable.exist(new_node.NodeID) then
9          return NodeAlreadyExistError
10     else
11         MTable.add(new_node.NodeID, new_node.Address)
12         return MarlinCommit ({SysLog})

13 Function DeleteNodeTxn(node_to_delete)
14     if MTable.exist(node_to_delete.NodeID) then
15         MTable.delete(node_to_delete.NodeID)
16         return MarlinCommit ({SysLog})
17     else
18         return NodeNotExistError

19 Function MigrationTxn(granule, src, dst)
       # Assume the transaction is triggered on the destination node
20     nid ← RPC^src_sync::GTable[granule].NodeID
21     if nid is src then
22         RPC^src_async::GTable[granule].NodeID = dst
23         GTable[granule].NodeID = dst
24         return MarlinCommit ({src, dst})
25     else
26         Abort and return WrongNodeError(nid)

27 Function RecoveryMigrTxn(granule, src, dst)
       # The transaction is triggered on the destination node
28     nid ← GTable[granule].NodeID
29     if nid is src then
30         GTable[granule].NodeID = dst
31     return MarlinCommit ({src.GLog, dst})

32 Function ScanGTableTxn()
33     nodes ← MTable.scan()
34     scan_res ← GTable.scan() asynchronously
35     for n in nodes do
36         if n is not cur_node then
37             scan_res.merge(RPC^n_async::GTable.scan())
38     return MarlinCommit ({SysLog} ∪ nodes)
```

2-3 and 6) and redirect the request to the correct owner. Moreover, compute nodes can periodically broadcast updates of their owned GTable partitions to routers, thereby reducing redirections.

Note that Marlin's correctness is independent of the isolation level used for user transactions. While user transactions must hold read locks on the corresponding GTable entries until they commit or abort, they can access user tables and operate at weaker isolation levels. For instance, if a user transaction employs Read Committed, the transaction could release a read lock on a user table right after the read. The long read lock on the GTable is orthogonal to the user table locks.

### 4.3 Consistency via MarlinCommit

Given that application data is strictly partitioned with each partition exclusively accessed by its owner compute node, conventional transaction protocols—such as concurrency control (e.g., 2PL [42], OCC [55]) and atomic commit (e.g., 2PC)—are sufficient to ensure ACID. Marlin leverages existing protocols in reference databases to maintain data consistency within each compute node.

However, relying solely on traditional protocols is insufficient in Marlin, where their core assumption—exclusive control of data by a single node—does not always hold. While Marlin minimizes cross-node consistency overhead by partitioning most states, some critical types of coordination states (i.e., MTable and GTable) must be accessible by multiple compute nodes to address the failover problem (recall §4.1). For instance, transactions modifying MTable (e.g., *AddNodeTxn* and *DeleteNodeTxn*) can execute concurrently on different nodes. Additionally, during compute node failures or network partitions, GTable partitions owned by failed or unhealthy nodes may be accessed simultaneously by *RecoveryMigrTxn* on multiple recovering nodes and by user transactions on the previously unhealthy node when it returns to normal.

This problem calls for additional mechanisms to ensure data consistency when the same data can be accessed by multiple nodes. Marlin addresses this challenge by *MarlinCommit*, an optimized atomic commit protocol that commits a transaction only if no cross-node modification has occurred since each node's last observed commit. Our key insight is to leverage a conditional append API provided by the log to detect whether any other node has appended to the log since the node's last append. We first introduce the required Log API, then explain *MarlinCommit* protocol in detail.

*4.3.1 Log APIs.* Conventional commit protocols use a basic Log API, *Append(updates)*, which simply appends transaction updates to the log's tail. Marlin takes advantage of an enhanced API of *Append(updates, LSN)*, a conditional append operation that succeeds only if the current log version matches an expected version. For convenience, we call it *Append@LSN*, which is implemented as a remote procedure call (RPC) to the disaggregated storage service by leveraging near-storage computation capabilities. Formally, *Append@LSN* is denoted as:

$$\textbf{(status, new\_lsn)} \leftarrow \boldsymbol{RPC^{log}_{sync/async}}\textbf{::}\boldsymbol{Append(updates,\ target\_lsn)}$$

The superscript *log* identifies the target log instance; choices include per-node GLog (denoted by node.GLog) or the global SysLog. *Append@LSN* succeeds only if the log's current LSN matches *target_lsn*. Each compute node maintains a local *H-LSN*, which is the highest LSN it has successfully appended, and uses *H-LSN* as the *target_lsn* when issuing *Append@LSN*. In cases where the target log is exclusively accessed by the node, *H-LSN* will align with the newest LSN of the log, allowing the operation to succeed. However, if another node has updated the log, the operation will fail because the current LSN of the log will exceed the *target_lsn*. In such scenarios, the newest LSN is returned to the caller, enabling it to retry the operation with an updated *target_lsn* if desired.

*Append@LSN* can be implemented on any storage system supporting compare-and-swap (CAS), a feature commonly supported by modern cloud-native storage services [21, 29, 40, 65, 68, 79]. For instance, many cloud storage services can provide CAS using entity tags (ETags) and conditional

**Algorithm 2: MarlinCommit** — Assume the coordinator is one of the participants. Difference between *MarlinCommit* and standard commit is highlighted in gray.

```
1  Function MarlinCommit(participants)
2      updates ← writes made by the txn in the current node
3      if len(participants) = 1 then
           # One-phase Commit if only the current node is involved
4          return TryLog(participants[0], updates)
5      else
           # Two-phase Commit if multiple participants are involved
6          for p in participants do
7              if p is a log instance then
8                  TryLog(p, {VOTE-YES} ∪ updates) asynchronously
9              else
                   # On receiving VOTE-REQ, node p votes and calls TryLog to log its vote and updates
10                 send VOTE-REQ to p asynchronously
11         return final decision according to responses of participants
12         while broadcasting the decision to all participants asynchronously

13 Function TryLog(log, updates)
14     (status, new_lsn) ← RPC^log_sync::Append(updates, lsn_tracker[log])
15     if status is FAILURE then
16         ClearMetaCache(log)
17         lsn_tracker[log] ← new_lsn
18         return ABORT
19     else
20         lsn_tracker[log] ← new_lsn
21         return COMMIT
```

headers such as *If-Match* [37–39]. When a client issues an *Append@LSN* request, the storage service checks whether the ETag on the log matches the ETag provided by the client. If they match, the updates are appended to the log and a new ETag is returned; otherwise, the operation fails and returns the current ETag instead. Further details can refer to §5.

*4.3.2 MarlinCommit.* Conventional commit protocols (e.g., 1PC, 2PC) focus on providing atomic commit guarantees for transactions. MarlinCommit modifies conventional protocols to expand their scope: it can ensure strong transactional guarantees even if multiple nodes can access the same data. MarlinCommit commits a transaction only if no cross-node modification has occurred since each node's last observed commit.

To achieve this, each node maintains a *lsn_tracker* array to track the last committed LSN *H-LSN* of each log in the cluster. *MarlinCommit* leverages the conditional append API *Append@LSN* to commit the transaction only when *H-LSN* tracked by the current node matches the newest LSN of the log. This ensures that no other node has modified the log during the transaction, thereby preventing cross-node inconsistency. The pseudocode in Algorithm 2 outlines this protocol.

***MarlinCommit(participants).*** After a transaction finishes the execution phase, the coordinator, normally the node that launches the transaction, calls *MarlinCommit(participants)* to start the atomic commit protocol. Depending on the number of participants, the protocol chooses between one-phase commit and two-phase commit (line 3). *MarlinCommit* generally follows the conventional

commit protocol but with two key differences. First, it replaces the conventional *Log()*, which directly appends the updates to the log, with *TryLog()* (lines 4, 8, 10), which conditionally appends the updates to the log if LSN matches or otherwise invalidates the caches of corresponding coordination states. Second, MarlinCommit does not limit participants to compute nodes; instead, a participant can be either a compute node or a log instance in the disaggregated storage. The rationale is twofold: (1) the log is the ground truth of the system, and (2) voting through a node is semantically equivalent to appending the vote directly to the log. This design allows a transaction to progress in the commit phase even if compute nodes fail, laying the foundation for failover transactions like *RecoveryMigrTxn* as presented in §4.2.

***TryLog(log, updates).*** This method is the core mechanism for managing the log. It appends the updates via *Append@LSN* (line 14) to the log, which succeeds only if the log's current LSN matches the node's last known LSN (*H-LSN*). If successful, it updates *H-LSN* and returns a `COMMIT` decision (lines 20-21). Otherwise, a cross-node modification on the same log may have occurred, so the transaction is aborted to prevent inconsistencies. Because the log's cache may now be outdated, it should be invalidated via *ClearMetaCache(log)* (line 16). Specifically, if the log is SysLog, the MTable cache is evicted, whereas for a node.GLog, the corresponding node's GTable cache is evicted. Since only coordination states can encounter cross-node modification, there is no need to evict the cache for user data. The compute node then updates *lsn_tracker* based on the latest LSN returned by *Append@LSN* (line 17) and returns an `ABORT` decision, indicating a transaction retry. The next transaction that encounters a cache miss in system tables will fetch the latest data from the storage layer via *GetPage@LSN*, guided by the updated *H-LSN*.

We note that Marlin does not have the blocking issue [9, 13, 84] that traditional 2PC protocols suffer from. The state-of-the-art 2PC protocol, Cornus [47], has solved this blocking issue by leveraging disaggregated storage: it allows an active compute node to append an `ABORT` to the log of an unresponsive participant, solving the blocking conservatively at the cost of aborting the transaction. This was possible because the disaggregated storage is highly-available and accessible by all compute nodes. Marlin can adopt this mechanism through its *TryLog()* primitive to avoid blocking in *user transactions*. Going further, *MarlinCommit* can append `VOTE-YES` to unresponsive nodes for *reconfiguration transactions*, guaranteeing progress for critical reconfigurations.

### 4.4 Coordination At Work

This section demonstrates how system tables and reconfiguration transactions in Marlin can be composed to achieve cluster management for cloud OTLP databases under typical autoscaling scenarios. We use *scale-out* and *failover* as two examples. Other scenarios, such as scale-in and rebalancing, can be implemented similarly.

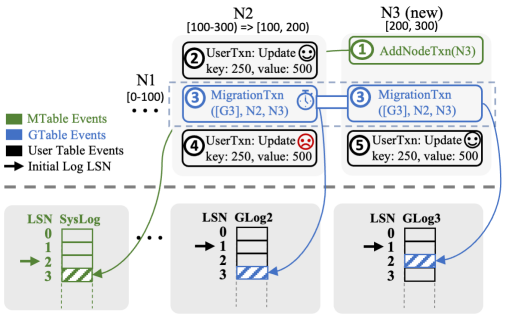



Fig. 6. Scale-Out Example.

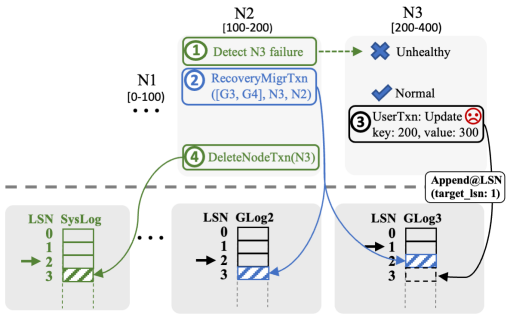



Fig. 7. Failover Example.

*4.4.1 Scale-Out.* When the user workload spikes, cloud databases will add more compute nodes to the cluster and spread the load evenly among the cluster. Figure 6 illustrates the scale-out process with a concrete example. Initially, node N2 manages the range [100, 300), including granules G2 ([100, 200)) and G3 ([200, 300)). After scale-out, a new node N3 takes over G3, reducing N2's range to [100, 200). Scale-out can be achieved in two steps: (1) adding the new node to the cluster and (2) rebalancing data from existing nodes to the new node. These steps are demonstrated in *Membership Updates* and *Live Migration*, respectively.

**Membership Update.** The new node triggers an *AddNodeTxn* to add itself to MTable (①). *AddNodeTxn* is a single node transaction that commits on SysLog at LSN 3. If conflicting *AddNodeTxn* and *DeleteNodeTxn* run in parallel, MarlinCommit ensures that only one transaction is committed. The new node may broadcast this membership change to all nodes after the transaction commits to reduce cache staleness. However, this is not required for correctness, as other nodes will discover this change when issuing their next transaction on MTable. MarlinCommit of the next transaction will discover that SysLog has changed because *Append@LSN* can no longer append to the log according to an outdated *H-LSN* saying 2. The node then invalidates its local cache for MTable and retries the transaction by fetching the newest data from the page store in the storage layer.

**Live Migration.** To redistribute load onto newly added nodes, the database transfers ownership of selected granules via a *MigrationTxn*. As shown in Figure 6, this transaction updates the relevant GTable partitions (P2 and P3) to move ownership of granule G3 from N2 to N3. It is a cross-node transaction (③) that commits on both the source (N2) and destination (N3): N2 logs its updates to GLog2 for GTable P2, while N3 logs its updates to GLog3 for GTable P3. Although a *MigrationTxn* can originate on either node, it usually starts on the underutilized destination node.

If the source node is running a user transaction, the transactional concurrency control protocol prevents conflicts. For instance, under 2PL, an ongoing user transaction on N2 (②) holds a write lock on G3, blocking the *MigrationTxn* from acquiring its required write lock until the user transaction commits. After ownership changes, any new user transaction arriving at N2 (④) aborts because it discovers that N2 no longer owns G3 during the data-effectiveness check. The aborted transaction can inform the issuing clients of ownership change, prompting them to redirect subsequent transactions on G3 to N3 (⑤). Because GTable is partitioned, *MigrationTxn* can run in parallel for different granules on different nodes, thus avoiding the scalability bottleneck of centralized coordination.

After migration, user transactions (⑤) on the newly added node may experience performance degradation due to a cold cache. Various live migration protocols have addressed this issue [35, 41]. Since Marlin focuses on coordination mechanisms, it can integrate with compatible protocols targeting similar reference cloud databases described in §3 by replacing their internal cluster coordination service. In our implementation, we mitigate the cold-cache issue by proactively warming up the cache after *MigrationTxn* updates ownership, following an approach similar to Squall [41]. Specifically, the destination node issues a scan query to the source node and populates its local cache with the scan results for uncached data.

*4.4.2 Failure and Recovery.* This section shows how Marlin supports failover without relying on an external coordination service. We begin by briefly outlining failure detection and then explain how to achieve consistent failover through a concrete example.

**Failure Detection.** Without a centralized coordination service, Marlin can use standard decentralized failure detection approaches such as ring-based heartbeating [12, 86], SWIM [34], or gossip-based heartbeating [43]. Marlin adopts a ring-based heartbeating approach akin to Orleans [12, 62] to monitor node health. Compute nodes in MTable form a ring (sorted by node ID)

and each node periodically sends heartbeat messages to its $k$ successors in the ring. If a successor fails to respond after a configurable number of attempts, the monitoring node assumes that the successor has failed and initiates a *Failover* procedure. This protocol can be further optimized to reduce false positives by letting compute nodes record "suspicious" votes for unresponsive nodes in MTable. A node is considered dead only when such votes exceed a threshold over a defined interval. Since these optimizations are orthogonal to our coordination mechanism, we leave them for future work.

**Failover.** Figure 7 demonstrates how Marlin achieves failover. Suppose node N3, which initially owns granules G3 and G4, becomes unhealthy due to a temporary slowdown but eventually recovers. The failover process progresses through several key steps:

First, when N2 detects a heartbeat timeout from N3 (①), it initiates a *RecoveryMigrTxn* to transfer G3 and G4 granules from N3 to N2 by updating GTable P2 and P3 (②). Although executed solely on N2, this transaction commits on both GLog2 at LSN 3 and GLog3 at LSN 2 through MarlinCommit. *RecoveryMigrTxn*'s ability to modify coordination states owned by N3 (GTable P3) while N3 is unresponsive is key to solving the failover problem - the failover can progress even when the owner of coordination states has failed. However, if not taken care of, data races might occur, as the same data could be modified simultaneously by multiple nodes (N2 and N3) if the failed node recovers. MarlinCommit can ensure correctness even in such race conditions by leveraging *Append@LSN*.

For example, when N3 recovers and attempts a user transaction to update G3 (③), the transaction aborts during MarlinCommit. Specifically, the conditional append operation *Append(updates, H-LSN)* on GLog3 fails because the current LSN of GLog3 has advanced to 2, while the *H-LSN* tracked by N3 remains at 1, causing a version mismatch. In response, N3 invokes *ClearMetaCache* to invalidate the cache for GTable P3. Upon refreshing its local cache of GTable P3, triggered by the next cache miss, N3 detects that it no longer owns G3 and G4. Thus, any ongoing or incoming transactions on N3 targeting these granules are aborted, preserving data integrity.

After migration, N2 runs *DeleteNodeTxn* (④) to remove N3, committing the change to SysLog. N2 may broadcast this membership change for quicker synchronization across nodes (akin to "Watch Notifications" in ZooKeeper [51]), although such a broadcast is not mandatory for correctness.

### 4.5 Correctness Invariants

We present the invariants maintained by Marlin's transactions to show their correctness. Central to Marlin's protocol is the Exclusive Granule Ownership invariant.

**I0: Exclusive Granule Ownership:** for any granule $G$ at any time, there is exactly one owner node $N$. This core invariant is a composition of the following sub-invariants:

- **I1: Reconfiguration Transactions are Serializable:** for any log, transactions on the log are serialized via the *TryLog* operation of *MarlinCommit*. Given this, we only need to consider each reconfiguration transaction individually.
- **I2: Node and GTable are 1-1 Mapped:** *AddNodeTxn*, *DeleteNodeTxn*, and *ScanGTableTxn* never update GLogs and are only involved during membership changes; their serialization makes SysLog the only source of membership, where each node is associated with exactly one GLog (hence one GTable).
- **I3: Owner Exists:** for each granule, there is always an owner, because a GTable can only be updated by a *MigrationTxn* or *RecoveryMigrTxn*, both swap a granule entry and never delete.
- **I4: Owner is Unique:** each granule has at most one owner, because both *MigrationTxn* and *RecoveryMigrTxn* swap the ownership of a granule entry and never end with dual owners.

These invariants jointly establish exclusive ownership. We provide the complete correctness proof and the TLA$^+$ specification in our technical report [93].

## 5 Implementation

We implement Marlin as a coordination mechanism on an OLTP database testbed[2]. Our testbed extends Sundial [98], an open-source distributed OLTP DBMS, into a storage-disaggregated DBMS following the system model as described in §3. The testbed architecture comprises three layers: clients, database servers, and the storage service. The client executes user transactions in interactive mode (in contrast to the stored procedure mode), where a new request is issued only after the previous response is received. The client communicates with the database server via gRPC [2], using either synchronous communication (e.g., data access requests) or asynchronous communication (e.g., two-phase commit messages). Exponential back-off is employed for aborted transactions.

Each database server contains a transaction manager and a cache manager. The transaction manager implements the standard protocols such as two-phase locking (2PL) for concurrency control and two-phase commit (2PC) for atomic commit. We leverage group commit to reduce the storage access overhead by batching log records from multiple transactions and committing them through a single log operation. By default, all transactions follow serializable isolation through the *NO_WAIT* protocol, which avoids deadlocks. The cache manager uses the clock replacement algorithm. On a read miss, the page is fetched from the disaggregated storage. If the requested data has a stale LSN, the storage node waits for log replay before replying. The storage layer contains a write-ahead log (WAL) for each compute node and a page store service. Following the log-as-database paradigm, committed transactions send only updates to the WAL, and data pages are asynchronously reconstructed via log replay. The page store uses Azure Table Storage [8], with all services hosted under a single *standard general-purpose v2* Azure storage account [66] in West US 2. WALs are stored in Azure Append Blobs [64], which support atomic and concurrent appends.

A key complication in our implementation is that the *MarlinCommit* protocol requires atomic conditional append (i.e., *Append@LSN*) from the storage service. We describe how we implemented it in Azure Blob Storage and how it can be realized in other cloud storage services.

**Microsoft Azure Blob Storage:** Azure's *append blobs* support atomic conditional appends via the *AppendBlock* operation, using precondition headers like *If-Match (ETag)* or *x-ms-blob-condition-appendpos-equal*. An LSN can be implemented using either ETag or blob length (the latter is currently used in Marlin). *AppendBlock* is called with one of these headers; the operation succeeds only if the condition holds. Otherwise, an error is returned, and the compute node can read the new ETag or length and retry. This check-and-append operation is guaranteed to be atomic within Azure Storage.

**Amazon S3:** S3 Express One Zone [5] can achieve atomic conditional appends using a single PUT with headers like `If-Match` or *x-amz-write-offset-bytes* as compare-and-swap primitives.

**Google Cloud Storage (GCS):** GCS assigns each object a monotonically increasing *generation number*, which can be used as LSN, and supports conditional writes via precondition headers like *ifGenerationMatch*. The client uploads data to a temporary object, then issues a *compose* operation that merges *log@<target_lsn>* with the temp object, using *ifGenerationMatch:<target_lsn>*.

Note that although Marlin is primarily designed for the Partitioned-Writer archetype, it is generalizable to other architectures discussed in §2.2. For both Single-Writer and Shared-Writer archetypes, the GTable is not needed since the data is not partitioned across multiple nodes—in Single-Writer only a single node can write to any data, and in Shared-Writer any node can write to any data. In these two archetypes, membership management can still follow Marlin's design via MTable and its associated reconfiguration transactions. Since most of the design complexity of Marlin is in the GTables, Marlin can be substantially simplified for these other two archetypes.

[2]https://github.com/CloudOLTP-UWM/Marlin/tree/sigmod26

## 6 Evaluation

We empirically evaluate the performance and cost of Marlin in comparison to external converged coordination services. Our key findings are as follows:

- Marlin outperforms ZooKeeper in both performance and cost efficiency in scale-out scenarios (§6.2 and §6.3).
- As coordination workloads scale, Marlin maintains superior performance and cost efficiency, while ZooKeeper and FoundationDB must trade off performance and cost efficiency ( §6.4).
- The performance and cost benefits of Marlin are further amplified in geo-distributed environments (§6.5) and preserve in more complicated and dynamic workloads (§6.6).
- Marlin's membership management performance is comparable to other baselines under moderate contention but may degrade under high contention ( §6.7).

### 6.1 Experimental Setup

*6.1.1* ***Configurations.*** We use the DBMS testbed described in §5. For a fair comparison, we implement Marlin and all baselines on this testbed. All experiments are conducted on Microsoft Azure Cloud Service platform [67]. Compute nodes are instances of Standard D4s v3 in the West US 2 region. Each runs on the 3rd Generation Intel(R) Xeon(R) Platinum 8370C processor with 4 vCPU, 16 GB memory, and a 2 Gbps network.

*6.1.2* ***Baselines.*** We compare Marlin against two external coordination services, ZooKeeper and FoundationDB. ZooKeeper represents an external and converged coordination solution. To better capture its scalability, we evaluate two ZooKeeper setups with different hardware configurations. FoundationDB is a distributed OLTP database offering high performance and scalability, and is used as a coordination service in several commercial cloud databases (e.g., Snowflake). Marlin and all baselines adopt the same cache-warming strategy during reconfiguration (§4.4.1).
**ZooKeeper-Small** (*S-ZK*): The baseline uses ZooKeeper 3.8.4 and JDK 12.0.2. The Zookeeper cluster consisted of one leader node and two follower nodes. Each node is a Standard D4s v3 instance with 4 vCPU, 16 GB memory, and 2 Gbps network.We set maxClientCnxns as 0 to disable client connection limits and set maximum heap memory of JVM as 15GB.
**ZooKeeper-Large** (*L-ZK*): This baseline uses a similar deployment as *S-ZK* except that each node has a larger configuration: Standard D8s v3 with 8 vCPU, 32 GB memory, and 4 Gbps network bandwidth. The maximum heap memory of JVM is 30GB.
**FoundationDB** (*FDB*): We deploy FoundationDB (7.3.63) on hardware comparable to *S-ZK*. Each node has one transaction process, one storage process, and one stateless process. The replication factor is three. Data is partitioned via a dynamic sharding mechanism based on key prefixes.

*6.1.3* ***Workloads.*** We use the YCSB and TPC-C OLTP workloads for performance evaluation.
**YCSB:** The Yahoo! Cloud Serving Benchmark (YCSB) [25] is a synthetic benchmark designed to evaluate large-scale Internet applications. We use tables with different sizes (ranging from 3 GB to 20 GB) that are partitioned into granules across servers by range on the primary key. Unless otherwise specified, each tuple is around 1KB and each granule is 64 KB. Each transaction is single-site and has 16 requests with 50% reads and 50% updates accessing 16 tuples. We generate requests following a uniform distribution. We use the YCSB workload as the default unless otherwise stated.
**TPC-C:** TPC-C [90] models a warehouse-centric order processing application with nine tables and five transaction types. All tables except ITEM are partitioned by the warehouse ID. The ITEM table is replicated at each server. 10% of NEW-ORDER and 15% of PAYMENT transactions access multiple warehouses; other transactions access data on a single server. We use a warehouse as the unit of migration, and each granule contains one warehouse. To evaluate performance under heavy

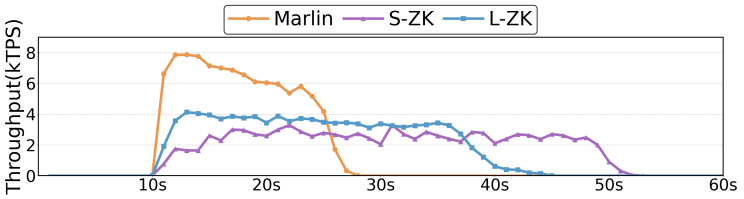


Fig. 8. MigrationTxn Throughput Over Time.

migration with a large number of warehouses, we tune down the size of each warehouse to ~1MB by reducing the number of customers per district, and deploy 1600 warehouses per server.

We design four testing scenarios to simulate real-world workloads. The **scale-out** scenario evaluates the system's ability to handle overloads by scaling out the cluster (§6.2–§6.4). The **geo-distributed** scenario models deployments where clients and compute nodes span multiple regions (§6.5). The **dynamic** scenario simulates bursty workloads with fluctuating client counts (§6.6). Finally, the **membership-update** scenario evaluates group membership management under intensive membership changes, such as cluster rescale or node failures and recoveries (§6.7).

*6.1.4* ***Methodology.*** We fully warm up the cache so that performance is stable before running experiments. We run the experiments for a fixed amount of time and report throughput and latency for committed transactions. If a transaction aborts, the client will keep retrying with exponential backoff (bounded by 100 ms) until it succeeds. In Marlin, each reconfiguration transaction migrates one granule, and multiple reconfiguration transactions are executed concurrently. The number of concurrent migration transactions is increased as the number of compute nodes increases.

*6.1.5* ***Cost Calculation.*** The total system cost includes data-plane and control-plane costs. *DB Cost* accounts for computing servers and cloud storage, while *Meta Cost* reflects coordination expenses. Since Marlin eliminates the external coordination service, its *Meta Cost* is zero. Computing server costs are calculated based on the machine's hourly rate. Storage costs are excluded from comparisons due to their negligible impact. For example, a single Standard D4s v3 instance costs $0.192 per hour, 384× higher than the cost of storing 20GB in the hot-tier blob storage.

## 6.2 Scale-Out Performance on YCSB

We evaluate the scale-out scenario using YCSB workload. The experiment runs a static workload of 800 concurrent clients accessing a 24 GB table partitioned into ~200K granules. Starting with 8 nodes, the cluster scales out to 16 nodes at the 10th second. This process executes ~100K *MigrationTxn*s, with each transaction migrating a single granule from an existing node to a newly added node.

Figure 8 shows that Marlin achieves 2.3× and 1.9× higher throughput for migration transactions than *S-ZK* and *L-ZK*, respectively. This gain stems from Marlin's distributed design, which partitions GTable across multiple nodes, spreading the metadata update load. In contrast, ZooKeeper relies on a centralized single-writer node, which becomes a bottleneck. The throughput advantage of *L-ZK* over *S-ZK* is due to *L-ZK*'s superior hardware, including higher network and disk bandwidth.

Marlin completes the scale-out process 2.6× and 1.9× faster than *S-ZK* and *L-ZK*, due to its higher migration throughput. As shown in Figure 9, the shorter migration duration promotes elasticity: the throughput of user transactions reaches a higher level of approximately 12k tps more rapidly than ZooKeeper-based approaches. Furthermore, Marlin has a lower abort rate for user transactions. This is because migration transactions in Marlin have lower latency and thus a lower chance of conflicting with user transactions. Figure 10a shows that Marlin reduces the migration latency by 2.57× and 1.87× compared to *S-ZK* and *L-ZK*. While each migration includes a cache warm-up phase, the latency of *MigrationTxn* remains the dominant factor of the performance difference.

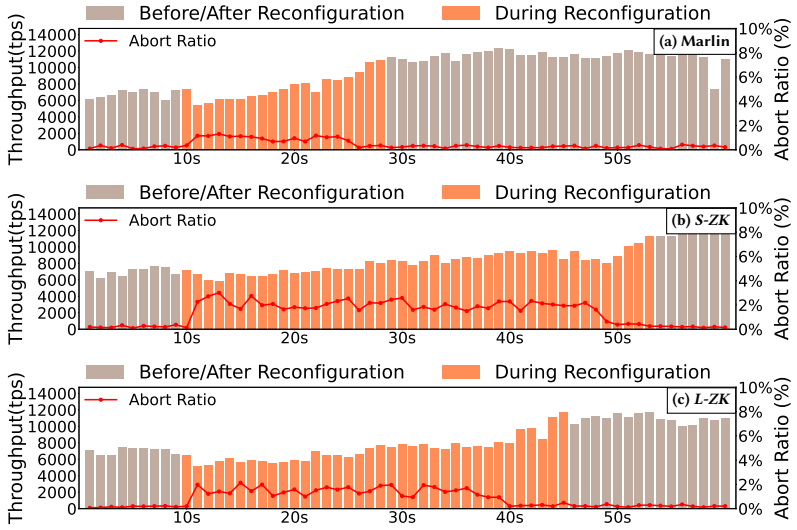
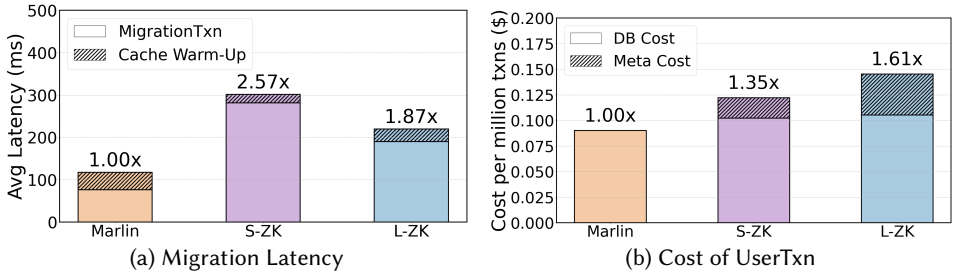



Fig. 9. Realtime Throughput of User Transactions (YCSB).




(a) Migration Latency (b) Cost of UserTxn

Fig. 10. Cost and Latency Breakdown.

Besides its performance benefits, Marlin also has the best cost efficiency. As shown in Figure 10b, Marlin reduces cost by 1.35× and 1.61× compared to *S-ZK* and *L-ZK*, respectively. This is primarily because Marlin eliminates the upfront cost of a static, converged coordination service, which requires a cluster of at least 3 nodes in ZooKeeper. The hourly rate for *S-ZK* and *L-ZK* cluster is $0.597 and $1.173, respectively. Furthermore, Marlin slightly decreases *DB Cost* per transaction by allowing the database to capitalize on scale-out benefits more quickly than ZooKeeper, thereby achieving higher throughput using the same *DB Cost*.

### 6.3 Scale-Out Performance on TPC-C

This section evaluates Marlin on the TPC-C benchmark. Similar to §6.2, the cluster initially consists of 8 nodes and scales out to 16 nodes at the 10th second. This process executes 6.4K *MigrationTxn*s, each migrating a single warehouse from an existing node to a newly added node. Each new node launches 80 migration threads to execute *MigrationTxn* concurrently.

As shown in Figure 11, Marlin completes the migration 2.5× and 1.5× faster than *S-ZK* and *L-ZK*, respectively. The improvement comes from Marlin's distributed coordination design that partitions GTable across multiple nodes, avoiding the centralized bottleneck inherent to ZooKeeper-based systems. The duration of migration in TPC-C is significantly shorter than in the YCSB experiments because the number of granules to migrate is smaller (6.4K vs. 200K). The performance gap between *L-ZK* and *S-ZK* is mainly due to the superior hardware configuration of *L-ZK*. Furthermore, Marlin incurs less degradation of user transactions during migration, as reflected in higher user throughput

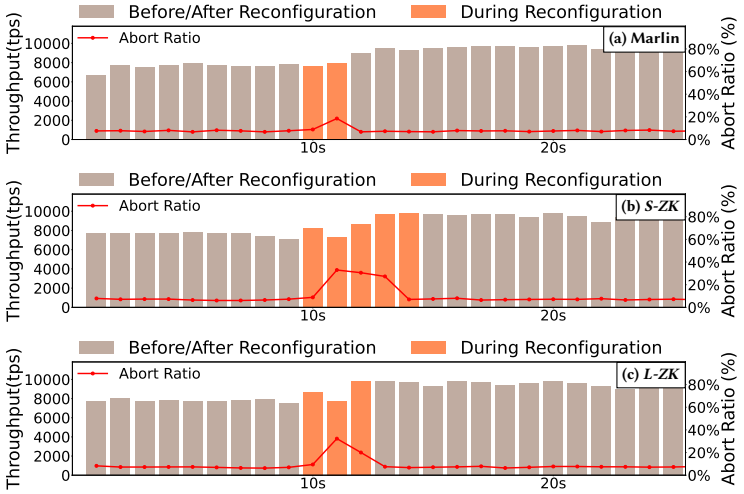
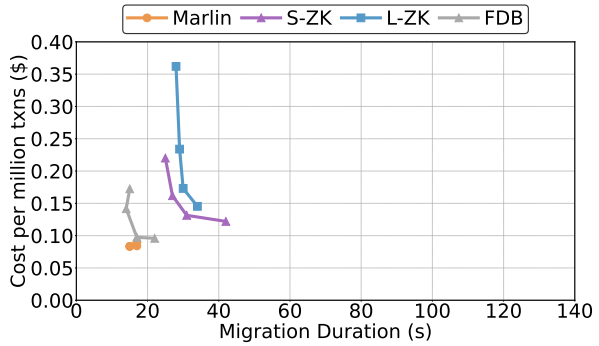
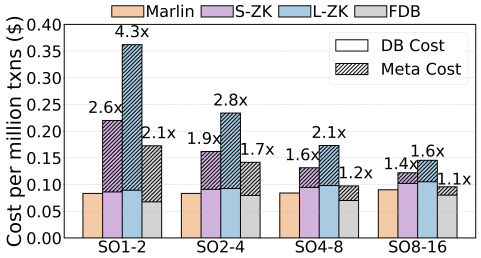


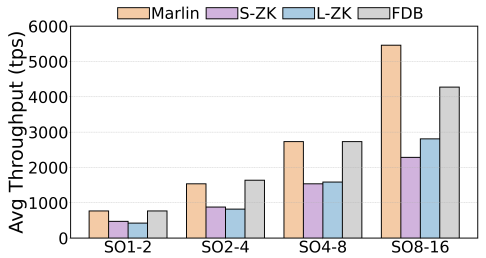

Fig. 11. Realtime Throughput of User Transactions (TPC-C).




(a) Cost per Transaction vs. Migration Duration




(b) Cost of UserTxn




(c) Migration Throughput

Figure 12. Cost vs. Migration Performance (Single-Region).

and lower abort rate. This improvement comes from lower-latency and higher-throughput migration transactions, which reduce conflicts with ongoing user transactions.

### 6.4 Cost vs. Migration Duration Tradeoff

We evaluate whether the benefits of Marlin persist across various scale-out scenarios with different data scales under YCSB workload. The four settings—*SO1-2*, *SO2-4*, *SO4-8*, and *SO8-16*—represent scale-outs from 1 to 2, 2 to 4, 4 to 8, and 8 to 16 nodes. These scenarios correspond to workloads of 100, 200, 400, and 800 clients, with table sizes of 3, 6, 12, and 24 GB. As the scale increases, the

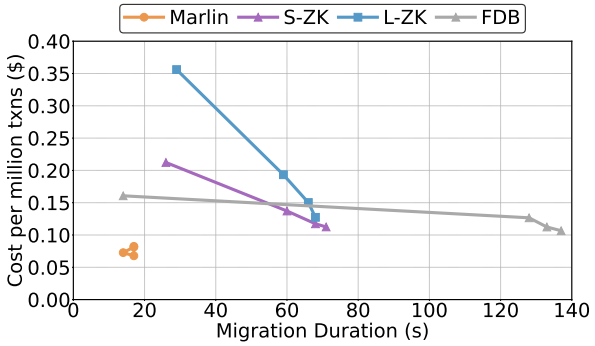



Figure 13. Cost vs. Migration Duration (Geo-Distributed).

number of migration transactions grow from ~10K to ~100K, with migration concurrency doubling at each step. In every data scale, the static workloads exceed the cluster's initial capacity, requiring a scale-out to double the cluster size to handle the workload effectively.

Figure 12a shows Marlin consistently achieves the best balance between cost and migration duration across all scales. It maintains the lowest cost per user transaction and shortest migration duration, with up to 4.4× lower cost than *L-ZK* in *SO1-2* and 2.5× faster migration than *S-ZK* in *SO8-16*. The cost-benefit is because Marlin eliminates the upfront cost for an external coordination service. The migration duration benefit is due to Marlin's partitioned design for GTable, which avoids centralized bottlenecks and therefore achieves higher migration throughput than ZooKeeper-based approaches. While *S-ZK* is more cost-effective than *L-ZK* due to its smaller hardware configuration, it causes longer migration durations compared to *L-ZK*, especially at large-scale migration (e.g., *SO8-16*) as it reaches its scalability limit. This is evident in Figure 12c, where the migration throughput in Marlin increases linearly with the intensity of migration workloads, while *S-ZK* shows exhibits diminishing gains, especially near the *SO8-16* scale.

Compared to *S-ZK* and *L-ZK*, *FDB* demonstrates shorter migration durations, benefiting from FoundationDB's internal partitioning that offers better scalability. However, *FDB*'s scalability remains constrained by its fixed resources, as it does not automatically rescale with the underlying database and requires manual reconfiguration. In contrast, Marlin 's scalability naturally grows with the scale of the coordinated database. This is evident in Figure 12c, where Marlin 's migration throughput increases linearly with database scale, while the throughput of ZooKeeper-based and *FDB* approaches continues to grow but at a diminishing rate. *FDB* incurs a higher cost per transaction (up to 2.1×) compared to Marlin, due to the overhead of the external coordination cluster.

For external coordination approaches, Figure 12b shows that *Meta Cost* constitutes a decreasing portion (e.g., from 75% to 28% in *L-ZK*) of the total system cost when cluster size increases from *SO1-2* to *SO8-16*. As the cluster scales, *Meta Cost* becomes less significant relative to *DB Cost*. Therefore, Marlin provides greater cost benefits under smaller workloads. However, when the cluster scales, the scalability advantage of Marlin becomes more pronounced, leading to shorter migration duration and better elasticity. Therefore, Marlin either achieves cost efficiency or provides better elasticity on all database scales, striking the best balance on cost and migration performance.

### 6.5 Performance in Geo-distributed Settings

Geo-distributed databases are common in the cloud, driven by the need for improved performance and compliance with regional regulatory requirements. These systems strategically position compute servers closer to clients across different geographical locations. In this experiment, we deploy clients and compute nodes across up to four regions: US West, Asia East, UK South, and Australia

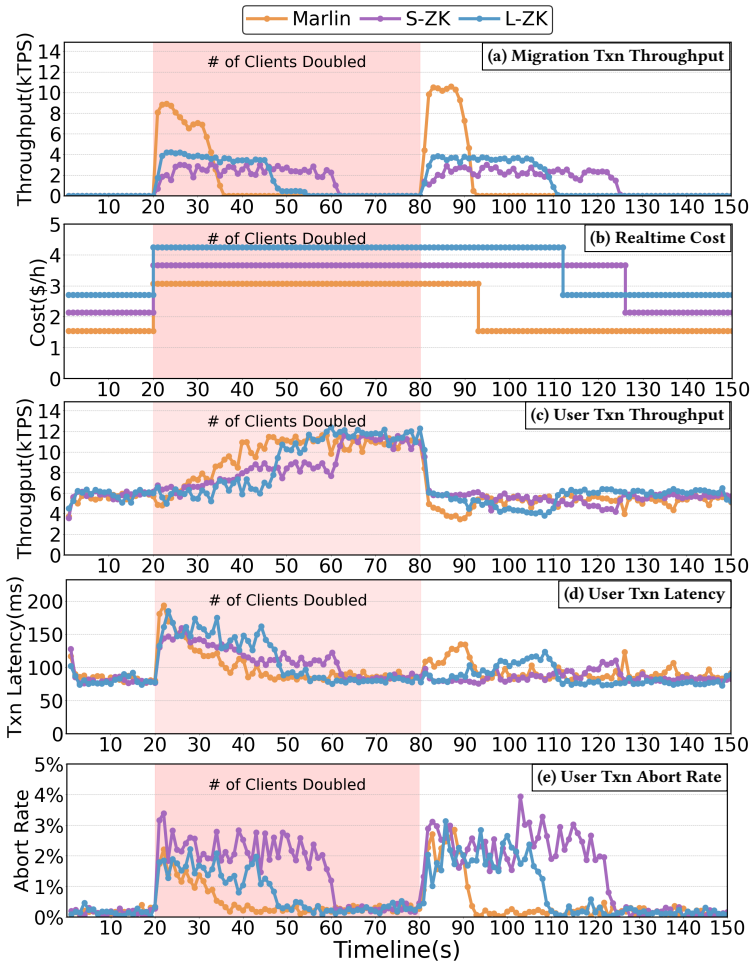



Fig. 14. Realtime Performance of Dynamic Workloads.

East. The workloads are the same as those in the scale-out scenarios in §6.4. Clients and compute nodes were evenly distributed across regions, with each client accessing only local compute nodes. The disaggregated storage service is co-located with its respective compute nodes. ZooKeeper and *FDB* are deployed in US West. For each scale-out, compute nodes within each region are doubled.

As illustrated in Figure 13, Marlin achieved up to 4.9× shorter migration duration than ZooKeeper-based methods and up to 9.5× shorter than *FDB* across all scales. This improvement is notably more pronounced in the geo-distributed setting, compared to the 2.5× gain observed in the single-region case in §6.4. The wider performance gap is due to the centralized nature of ZooKeeper-based approaches, which incur higher latency from cross-region communication. In contrast, Marlin 's distributed metadata management allows each region to manage its GTable partition independently, inherently co-locating coordination with compute and eliminating cross-region communication during migration. Interestingly, *L-ZK*'s migration duration was similar to *S-ZK* at all scales, as its hardware advantage was offset by cross-region latency. While *FDB* scales better than ZooKeeper, it incurs significantly longer migration times because each migration triggers a metadata update in *FDB*, requiring multiple cross-region round trips (e.g., *GetReadVersion* and commit requests), whereas ZooKeeper-based methods require only one. Regarding cost, Marlin remains the most

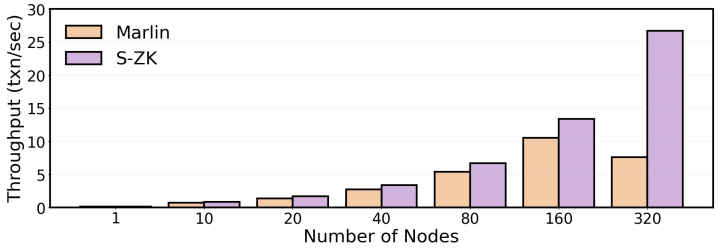



Figure 15. MTable Stress Test.

cost-efficient by eliminating a dedicated coordination service. The extent of Marlin 's cost advantage remains consistent with single-region deployments.

While ZooKeeper-based approaches and *FDB* can deploy one coordination cluster in each region to reduce cross-region latency, this increases the number of coordination clusters, leading to higher costs and operational complexity. For example, placing a ZooKeeper cluster in every region can raise the *Meta Cost* by up to 4× compared to the current setup. In contrast, Marlin co-locates coordination with compute nodes by design, avoiding additional cost overhead. These benefits make Marlin particularly effective in geo-distributed settings.

### 6.6 Dynamic Workloads

This experiment simulates a bursty workload scenario. The workload starts with 400 clients, scales to 800 at the 20th second, holds for 60 seconds, and drops back to 400 at the 80th second. The cluster begins with 8 compute nodes, scales out to 16, then returns to 8. An efficient coordination mechanism enables rapid scale-out and scale-in.

**Performance.** As shown in Figure 14a, Marlin achieves significantly higher migration throughput than ZooKeeper-based approaches, completing scale-out 2.6× and 2.3× faster than *S-ZK* and *L-ZK*, respectively, and scale-in 3.8× and 2.6× faster. Rapid scaling enables the database to reach higher user transaction throughput (~12k TPS) more quickly, as seen in Figure 14c. Furthermore, Marlin minimizes the performance impact of reconfigurations on user transactions. This can be evident by Figure 14c and Figure 14d: the latency and abort rates of user transactions return to normal levels faster than ZooKeeper-based approaches.

**Cost efficiency.** As shown in Figure 14b, Marlin incurs lowest real-time cost under the dynamic workload. This benefit comes from two aspects. ❶ Marlin avoids the upfront cost of external coordination cluster. From 0th to 20th second, Marlin reduces the cost by 1.4× and 1.8× compared to *S-ZK* and *L-ZK*. ❷ The rapid scale-in releases compute nodes more promptly, improving resource utilization and cost efficiency. Marlin reduces compute nodes 12 seconds later after the user workload drops, while *S-ZK* and *L-ZK* reduce compute nodes after 45 seconds and 32 seconds.

### 6.7 Membership Change Performance

Previous experiments primarily investigated migration performance, limited to updates on GTable. This experiment specifically evaluates the performance of group membership updates under *intensive membership changes*, effectively stress-testing MTable. To reflect realistic loads, we simulate each compute node with one thread continuously issuing membership update requests, including node additions and removals. We scale the number of nodes by increasing threads number. Each thread issues a membership update every 15 seconds, similar to the monitoring interval utilized by autoscaling systems to determine the necessity for reconfiguration.

As shown in Figure 15, Marlin performs comparably to ZooKeeper-based approaches up to 160 nodes. Beyond that point, performance degrades due to the overhead of optimistic concurrency control in the TryLog() API for SysLog, which incurs retries under high contention. This performance disparity is independent of our decision to integrate the coordination service into Marlin.

In practice, group membership changes are infrequent and initiated by a small subset of nodes rather than the entire cluster simultaneously, resulting in much lower intensity than our stress test. Thus, Marlin's membership change performance is likely sufficient in real-world scenarios. Moreover, established techniques for improving optimistic concurrency control under high contention (e.g., [19, 52, 61, 81, 97]) could be incorporated into Marlin, which we leave for future work.

## 7 Related Work

Existing database systems adopt one of the following three approaches to fault-tolerant metadata coordination. We discuss their relevance and difference with Marlin in this section.

**Single-master coordination services.** Prevalent converged coordination services take a single-master architecture, including lock services such as Chubby [16] and general configuration stores such as Apache ZooKeeper [51], Doozer [48], etcd [1], and FireScroll [80]. They are internally replicated for fault-tolerance and backed by an atomic broadcast or consensus protocol, e.g., ZAB [53], Paxos [57], or Raft [74], for consistency of the ordering of operations. Such coordination services introduce cost overheads and deliver suboptimal performance under autoscaling workloads.

**Sharded coordination services.** Some works divide coordination metadata into multiple shards and assign each shard to a separate consensus group, each with an independent master node. This approach is equivalent to deploying multiple coordination service clusters to balance the load of update requests. Consul [49] is a primary example of this approach that offers little cross-shard consistency guarantees. ZooNet [59] stitches the consistency of metadata operations between shards by fixing one coordination service cluster per datacenter and adding remote learners. Zelos [3] is a sharded ZooKeeper API atop Delos, a shared log system [11]. Other examples include the rebuilt ZooKeeper in Clickhouse [22] and the transactional metadata key-value stores in Snowflake [33] and ByConity [17]. Sharded services improve metadata scalability but have three significant drawbacks. First, they risk weakening metadata consistency for cross-shard operations, which are common in autoscaling migrations. Second, they dedicate even more resources to metadata coordination, enlarging the cost overhead. Third, they increase system architecture complexity.

**Coordination without a dedicated service.** Some cloud systems have explored techniques to facilitate metadata coordination without a dedicated service. KRaft [24] proposes a simplified architecture for Kafka, a consistent messaging service [54], by replacing ZooKeeper with embedded metadata quorums. FaasKeeper [26] is a serverless variant of ZooKeeper, built entirely upon on-demand serverless functions and shared disaggregated storage.

## 8 Conclusion

We propose Marlin, a cloud-native coordination mechanism for autoscaling databases with storage disaggregation. By integrating coordination within the database, Marlin achieves linear scalability, cost efficiency, and operational simplicity. It enables safe failover without external coordination services by permitting cross-node updates on coordination state, and introduces MarlinCommit, an optimized commit protocol, to ensure strong transactional guarantees under race conditions.

## Acknowledgments

We thank Xiaosuo Chen, Zhuangtianyu Liao, Yucheng Lu, Changchun Zhang, Wenqi Ma, and Cheng Lyu for their valuable contributions during the revision of this manuscript. We thank Xi Pang and the anonymous reviewers for their constructive suggestions. This work was supported in part by NSF award IIS-2144588.

## References


[1] 2024. etcd. https://github.com/etcd-io/etcd. Accessed: 2024-01-01.

[2] 2024. gRPC. https://grpc.io/. Accessed: 2024-01-01.

[3] Ali Zaveri and Suyog Mapara and David Devecsery. 2022. Introducing Zelos: A ZooKeeper API leveraging Delos. https://engineering.fb.com/2022/06/08/developer-tools/zelos/. Accessed: 2024-10-16.

[4] Amazon Web Services. 2023. *Amazon Redshift Spectrum.* https://aws.amazon.com/redshift/redshift-spectrum/ Accessed: 2024-11-13.

[5] Amazon Web Services. 2023. *Amazon S3 Express One Zone Storage Class.* https://aws.amazon.com/s3/storage-classes/express-one-zone/ Accessed: 2025-05-02.

[6] Amazon Web Services. 2023. *AQUA for Amazon Redshift.* https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift// Accessed: 2024-11-13.

[7] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[8] Microsoft Azure. 2025. Azure Table Storage. Online. https://azure.microsoft.com/en-us/products/storage/tables Accessed: 2025-01-15.

[9] Ozalp Babaoglu and Sam Toueg. 1993. Understanding non-blocking atomic commitment. In *Distributed systems*.

[10] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. 2020. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 617–632.

[11] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[12] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. Orleans: Distributed virtual actors for programmability and scalability. In *MSR-TR-2014–41*.

[13] Philip A Bernstein. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley New York.

[14] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder-A Transactional Record Manager for Shared Flash.. In *CIDR*, Vol. 11. 9–20.

[15] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 251–264.

[16] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 335–350.

[17] ByConity. 2024. ByConity: an open source cloud data warehouse. https://github.com/ByConity/ByConity. Accessed: 2024-10-16.

[18] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. {POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*. 29–41.

[19] M. J. Carey. 1987. Improving the performance of an optimistic concurrency control algorithm through timestamps and versions. *IEEE Transactions on Software Engineering* 13, 6 (1987), 746–751.

[20] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 398–407.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[22] ClickHouse, Inc. 2016. ClickHouse: Fast Open-Source OLAP DBMS. https://clickhouse.com/. Accessed: 2024-10-16.

[23] Cockroach Labs. 2024. CockroachDB. https://www.cockroachlabs.com/. Accessed: 2024-01-01.

[24] Colin McCabe. 2020. KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum. https://cwiki.apache.org/confluence/display/KAFKA/KIP-500. Accessed: 2024-10-16.

[25] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[26] Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, Konstantin Taranov, and Torsten Hoefler. 2024. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24)*. ACM, 94–108. doi:10.1145/3625549.3658661

[27] James C. Corbett and et al. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*. 251–264.

[28] IBM Corporation. 2024. *IBM DB2 Data Sharing*. https://www.ibm.com/docs/en/db2-for-zos/12?topic=db2-data-sharing IBM Documentation.
[29] Microsoft Corporation. 2024. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db Accessed: 2024-10-29.
[30] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.
[31] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed postgresql for data-intensive applications. In *Proceedings of the 2021 International Conference on Management of Data*. 2490–2502.
[32] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).
[33] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*.
[34] Abhinandan Das, Indranil Gupta, and Ashish Motivala. 2002. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 303–312.
[35] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.
[36] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. 2020. Taurus database: How to be fast, available, and frugal in the cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1463–1478.
[37] Amazon Web Services Documentation. 2025. *Amazon S3 ETag Overview*. https://docs.aws.amazon.com/AmazonS3/latest/API/API_Object.html Accessed: 2025-01-12.
[38] Google Cloud Documentation. 2025. *Google Cloud Storage ETag Usage*. https://cloud.google.com/secret-manager/docs/etags Accessed: 2025-01-12.
[39] Microsoft Documentation. 2025. *Optimistic concurrency in Azure Storage with ETags*. https://learn.microsoft.com/en-us/azure/storage/blobs/concurrency-manage Accessed: 2025-01-12.
[40] Mostafa Elhemali, Niall Gallagher, Bin Tang, Nick Gordon, Hao Huang, Haibo Chen, Joseph Idziorek, Mengtian Wang, Richard Krog, Zongpeng Zhu, et al. 2022. Amazon {DynamoDB}: A Scalable, Predictably Performant, and Fully Managed {NoSQL} Database Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 1037–1048.
[41] Aaron J Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 299–313.
[42] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (1976), 624–633.
[43] Ayalvadi J Ganesh, A-M Kermarrec, and Laurent Massoulié. 2003. Peer-to-peer membership management for gossip-based protocols. *IEEE transactions on computers* 52, 2 (2003), 139–149.
[44] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. Workload analysis and demand prediction of enterprise data center applications. In *2007 IEEE 10th International Symposium on Workload Characterization*. IEEE, 171–180.
[45] Google Cloud. 2023. AlloyDB for PostgreSQL. https://cloud.google.com/alloydb Accessed: 2024-11-13.
[46] The PostgreSQL Global Development Group. 2023. PostgreSQL: The world's most advanced open source relational database. https://www.postgresql.org/ Accessed: 2024-10-29.
[47] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. 2022. Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation. *Proc. VLDB Endow.* 16, 2 (2022), 379–392. doi:10.14778/3565816.3565837
[48] ha. 2011. Doozer. https://github.com/ha/doozerd. Accessed: 2024-10-16.
[49] HashiCorp. 2020. Consul. https://www.consul.io. Accessed: 2024-10-16.
[50] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[51] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.
[52] Theo Härder. 1984. Observations on optimistic concurrency control schemes. *Information Systems* 9, 2 (1984), 111–120.

[53] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*. 245–256. doi:10.1109/DSN.2011.5958223

[54] Jay Kreps. 2011. Kafka : a Distributed Messaging System for Log Processing. https://api.semanticscholar.org/CorpusID:18534081

[55] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[56] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[57] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[58] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. 2019. Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1453–1465.

[59] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2016. Modular Composition of Coordination Services. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 251–264. https://www.usenix.org/conference/atc16/technical-sessions/presentation/lev-ari

[60] Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fabio Porto, and Patrick Valduriez. 2013. Dynamic workload-based partitioning algorithms for continuously growing databases. *Transactions on Large-Scale Data-and Knowledge-Centered Systems XII* (2013), 105–128.

[61] D. A. Menascé and T. Nakanishi. 1982. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems* 7, 1 (1982), 13–27.

[62] Microsoft. n.d.. Cluster Management - Orleans Documentation. https://learn.microsoft.com/en-us/dotnet/orleans/implementation/cluster-management. Accessed: 2025-01-13.

[63] Microsoft Azure. 2023. *Azure SQL Database Hyperscale.* https://azure.microsoft.com/en-us/products/azure-sql/database/hyperscale/ Accessed: 2024-11-13.

[64] Microsoft Azure. 2024. Azure Append Blobs. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction#append-blobs. Accessed: 2024-01-01.

[65] Microsoft Azure. 2024. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/. Accessed: 2024-01-01.

[66] Microsoft Azure. 2024. Azure Storage Account Standard General-Purpose v2. https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview#general-purpose-v2-accounts. Accessed: 2024-01-01.

[67] Microsoft Azure. 2024. Microsoft Azure Cloud Service Platform. https://azure.microsoft.com/en-us/overview/. Accessed: 2024-01-01.

[68] Microsoft Corporation. [n. d.]. Azure Table Storage. https://azure.microsoft.com/en-us/services/storage/tables/. Accessed: 2023-10-29.

[69] Microsoft Corporation. 2024. Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/

[70] C Mohan and Inderpal Narang. 1991. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the 17th International Conference on Very Large Data Bases*. 193–207.

[71] Neo4j, Inc. 2025. Neo4j: The World's Leading Graph Database. https://neo4j.com Accessed: 2025-01-14.

[72] Neon Technologies. 2023. *Neon: Serverless Postgres.* https://neon.tech/ Accessed: 2024-11-13.

[73] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[74] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[75] Oracle. 2024. Oracle RAC. https://www.oracle.com/database/real-application-clusters/.

[76] Xi Pang and Jianguo Wang. 2024. Understanding the performance implications of the design principles in storage-disaggregated databases. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[77] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling highly contended OLTP workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.

[78] Erhard Rahm. 1989. *Recovery concepts for data sharing systems.* Citeseer.

[79] Redis. 2024. Redis. http://redis.io/. Accessed: 2024-01-01.

[80] Redpanda Data Inc. 2023. FireScroll. https://github.com/FireScroll/FireScroll. Accessed: 2024-10-16.

[81] P. L. Reiher and G. J. Popek. 1988. Optimistic concurrency control by predictive optimistic concurrency control. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*. ACM, 81–88.

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

[83] Zhiming Shen, Qin Jia, Gur-Eyal Sela, Ben Rainero, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. 2016. Follow the sun through the clouds: Application migration for geographically shifting workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 141–154.

[84] Dale Skeen. 1981. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*. 133–142.

[85] Benjamin Treynor Sloss, Mike Dahlin, Vivek Rau, and Betsy Beyer. 2017. The Calculus of Service Availability: You're only as available as the sum of your dependencies. *Queue* 15, 2 (2017), 49–67.

[86] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking* 11, 1 (2003), 17–32.

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

[88] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[89] ScyllaDB Team. 2024. ScyllaDB. https://www.scylladb.com/. Accessed: 2024-01-01.

[90] Transaction Processing Performance Council (TPC). 1992. TPC Benchmark C (TPC-C): Standard Specification. http://www.tpc.org/tpcc/. Accessed: 2025-03-07.

[91] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[92] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. 2009. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 conference on USENIX Annual technical conference*. 28–28.

[93] Wenjie Hu, Guanzhou Hu, Mahesh Balakrishnan, Xiangyao Yu. 2025. Marlin: Efficient Coordination for Autoscaling Cloud DBMS (Extended Version). https://arxiv.org/abs/2508.01931.

[94] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 157–168.

[95] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a CloudDBMS. In *VLDB*.

[96] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[97] Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. ACM, 1629–1642.

[98] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: harmonizing concurrency control and caching in a distributed OLTP database management system. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1289–1302.

[99] Yugabyte, Inc. 2023. YugabyteDB: A High-Performance Distributed SQL Database for Global, Internet-Scale Applications. https://www.yugabyte.com. Accessed: 2024-11-13.

[100] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655* (2016).

[101] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*. 2653–2666.

[102] Tobias Ziegler, Philip A Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem?. In *CIDR*.

[103] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data*. 685–699.

Received January 2025; revised April 2025; accepted May 2025