Amazon MemoryDB: A Fast and Durable Memory-First Cloud Database

Yacine Taleb
Amazon Web Services
Canada

Kevin McGehee
Amazon Web Services
USA

Nan Yan
Amazon Web Services
Canada

Shawn Wang
Amazon Web Services
USA

Stefan C. Müller
Amazon Web Services
Canada

Allen Samuels
Amazon Web Services
USA

Abstract

Amazon MemoryDB for Redis is a database service designed for 11 9s of durability with in-memory performance. In this paper, we describe the architecture of MemoryDB and how we leverage open-source Redis, a popular data structure store, to build an enterprise-grade cloud database. MemoryDB offloads durability concerns to a separate low-latency, durable transaction log service, allowing us to scale performance, availability, and durability independently from the in-memory execution engine. We describe how, using this architecture, we are able to remain fully compatible with Redis, while providing single-digit millisecond write and microsecond-scale read latencies, strong consistency, and high availability. MemoryDB launched in 2021.

CCS Concepts

• **Information systems** → **Main memory engines**.

ACM Reference Format:
Yacine Taleb, Kevin McGehee, Nan Yan, Shawn Wang, Stefan C. Müller, and Allen Samuels. 2024. Amazon MemoryDB: A Fast and Durable Memory-First Cloud Database. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626246.3653380

1 Introduction

For many real-time applications such as Finance, Advertising, and Internet-of-Things (IoT) applications, fast response time is critical, even more so when these applications require multiple consecutive data accesses. Modern key-value stores can provide millions of operations per second per machine and microsecond scale latencies [15, 21] with simple key-value semantics. However, these simple semantics often lead to additional complexity on clients and overhead on the storage system. For example, a real-time bidding application will often need to access multiple keys representing user profiles to perform real-time aggregations such as sorting profiles based on certain criteria. With billions of users, this does not scale and leads to application business logic being customized around the limitations of the underlying storage system.

Open Source Software (OSS) Redis [12], hereafter referred to as Redis, emerged as the most popular in-memory key-value store according to *db-engines.com*[5]. Redis provides microsecond latencies, with p99 under 400us [9]), while allowing applications to manipulate remote data structures, perform complex operations, and push compute to storage. Redis support for complex shared data structures substantially simplifies distributed applications and is chiefly responsible for its popularity. Redis employs asynchronous replication for high availability and read scaling and an on-disk transaction log for local durability. Redis does not offer a replication solution that can tolerate the loss of nodes without data loss, or can offer scalable strongly-consistent reads. This limits its ability to be leveraged for use cases beyond caching.

AWS supports Redis for caching with Amazon ElastiCache [1]. Many ElastiCache for Redis customers use Redis as their main data store for low-latency microservices applications. To work around Redis lack of durability, they build complex pipelines to ingest data, store it durably, then hydrate that data in Redis. When data loss is detected in Redis, a separate job must re-hydrate the cache. This adds significant complexity and cost and impacts availability. For example, a catalog microservice in an e-commerce shopping application may want to fetch item details from Redis to serve millions of page views per second. In an optimal setup, the service stores all data in Redis, but instead must use a data pipeline to ingest catalog data into a separate database, like Amazon DynamoDB [24], before triggering writes to Redis through a DynamoDB stream. When the service detects that an item is missing in Redis—a sign of data loss—a separate job must reconcile Redis against DynamoDB. Our customers find this approach complex, and asked for a way to simplify their architectures, reduce costs, and achieve better performance. We built MemoryDB for Redis to meet these needs.

MemoryDB is a fully managed in-memory cloud-based database service. We built MemoryDB to provide in-memory performance with durability, strong consistency, and high availability. MemoryDB ensures strong consistency with 99.99% availability while providing microsecond read and single-digit millisecond write latencies. MemoryDB achieves that by leveraging Redis as an in-memory execution engine and offloading data persistence to an internal scale-out transaction log service.

In this paper, we describe the architecture of MemoryDB and the lessons we learned while building and operating it, specifically:

- How MemoryDB leverages separation of concerns to provide durability, consistency, performance, and availability

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3653380

while remaining fully compatible with Redis, one of the most popular in-memory database engines.
- Mechanisms we developed to provide predictable performance at scale while ensuring correctness.

2 Background and Motivation

2.1 Background on Redis

Redis is an in-memory data-structure store with a rich feature set. This paper refers to OSS Redis [12] (version 7.0.7) as Redis throughout, and not to Redis Enterprise or Cloud offerings by Redis Inc [11]. Redis supports over 200 commands on 10 data structures, including hash tables, sorted sets, streams, and hyperloglogs [12]. Commands can be combined into atomic groups with all or nothing guarantees. Redis also supports server-side execution of Lua scripts which also execute atomically. The Lua scripting language is Turing-complete, allowing customers to implement complex logic wholly within the server, eliminating network round trips and complicated locking and synchronization mechanisms.

Redis [10] supports horizontal scaling. In this configuration, Redis splits its flat key space into 16384 slots using CRC16 [10]. These slots are distributed across one or more shards as part of server-side setup. Each shard has a single writer (the "primary") and zero or more read-only replica nodes. Redis clients discover the slot-to-shard mapping from any node in the cluster and send commands directly to a node owning a requested key. For maximum performance, Redis assigns the responsibility for routing of requests among cluster members to the clients themselves. If the mapping changes, for example after scaling, clients receive redirection instructions when attempting to execute a command on a key that is not owned by the node receiving the command. In this configuration, transactions involving multiple keys are supported as long as all keys belong to the same slot, ensuring that the transaction can be executed fully within one shard.

MemoryDB supports an atomic slot-level migration process (resharding), allowing nodes to continue servicing requests normally while performing migration. We describe the MemoryDB resharding in more details in 5.2.

Replication between primary and replica nodes is implemented as passive logical replication: mutating commands are first executed on the primary node, updating its data structures, and then asynchronously replicated by sending the command to the replicas. Reads from a replica are allowed and provide a consistent view of the data but at a past point in time. The time lag between the primary and replica nodes is not controlled by Redis and can become significant depending on system conditions. For example, the lag between a primary and its replica could increase if a replica is slow (e.g., overloaded) in consuming updates. Clients must explicitly opt-in to be able to read from replicas, ensuring that they do not accidentally consume stale data.

Not all Redis commands can be naively forwarded to replicas for execution, such as the SPOP command, which removes a random element from a set. When executing this command on a primary node, an element in the set is randomly selected. An explicit delete command for the selected element is then sent to replicas over the replication channel. Following the same pattern, when a Lua script (similar to a stored procedure in a traditional RDBMS) is executed, the script itself is not replicated, but the effects it had on the data set are recorded and replicated atomically. This mechanism allows non-deterministic operations to be replicated deterministically.

2.2 Challenges of Maintaining Durability and Consistency in Redis

2.2.1 Data Loss During Failover Besides serving read-only requests, replica nodes increase availability as they can be promoted to the role of primary within a shard if a primary node fails. Redis uses a quorum-based approach for both failure detection and election of new primaries. Since replicas have a shard's data available in-memory, write availability can be restored within seconds after a failure. However, the Redis quorum-based protocol does not guarantee consistent replica promotion as replication between primary and replica nodes is asynchronous. As a result, a failover will cause a permanent loss of the writes that were not replicated to the promoted node at the time of failure.

Redis implements a few mechanisms for lightweight persistence: point-in-time snapshots, and an on-disk transaction log. For snapshots, Redis implements an approach to generate a point-in-time snapshot by serializing all items to disk. Redis can also persist mutations to disk using an Append-Only File (AOF) feature that appends all mutating commands to a file. In the most conservative mode, AOF issues an *fsync()* for every update, which would synchronously flush to disk, effectively linearizing Redis. In a configuration with a single primary node, AOF could provide durability (as long as AOF file is available) at the expense of availability. In realistic configurations with multiple nodes, the failure of a primary node will trigger a leader election, trying to promote the replica that is most up-to-date with the failed primary. There is no mechanism to ensure the elected replica received all acknowledged updates, which can cause data loss. In the worst case, Redis could elect a replica with no data, causing all nodes within a shard to synchronize with it, leading to complete data loss.

2.2.2 Asynchronous Replication Redis provides a facility to enforce synchronous replication for a given client through the WAIT command. When a client issues WAIT to a given shard, it is blocked until all prior updates that have been executed are acknowledged by a configurable number of replicas. However, this does not prevent other clients accessing the same shard to observe unacknowledged updates as WAIT does not synchronize replication globally on a shard. Furthermore, in the event of a failover, there is no mechanism to enforce the promotion of a replica that observed all acknowledged writes.

At Amazon, we design for failures, and although Redis does a good job at maintaining high availability, it can lose data. Our customers asked us for a solution to allow Redis to be used as a primary database with multi-Availability Zone (AZ) [1] durability. Our team faced the challenge of how to maintain consistency across all failure modes in Redis while minimizing both performance impact and divergence from the Redis code base so that we can support full parity with the Redis API now and into the future.

[1] An Availability Zone (AZ) is a subset of a Region that is connected to other AZs in the region through low latency links but is isolated for faults, including power, networking, software deployments, flooding, etc.

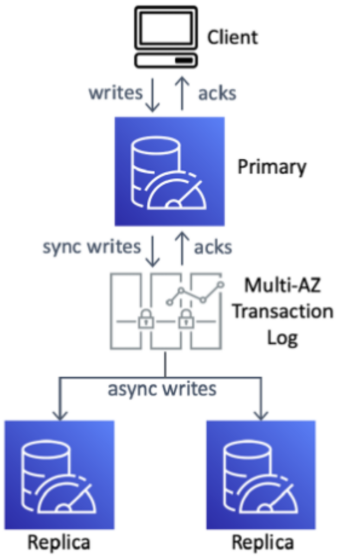


Figure 1: MemoryDB high-level overview. A primary node synchronously executes and replicates updates to a MultiAZ transaction log. Secondary nodes asynchronously fetch updates from the transaction log.

3 Durability and Consistency

A durable database system must ensure that once data is committed and acknowledged it can be read back. Common logging and replication strategies provide durability levels that are usually a factor of the number of available database cluster nodes and their (storage) lifetime [35]. However, nodes can fail, get terminated, or scaled, therefore decoupling durability from the database nodes can help in providing consistent durability guarantees. MemoryDB offloads durability by leveraging a distributed transaction log service. A transaction log provides low-latency and strongly consistent commits across multiple AZs.

3.1 Decoupling Durability

To minimize divergence from Redis, we followed a similar approach to Amazon Aurora [35]: we decomposed the stack into multiple layers, decoupling the execution engine from the durability layer. We use Redis as an in-memory execution and storage engine but redirect its existing replication stream into the transaction log, which is responsible for propagation of writes to replicas and leader election. This allows us to offer the full Redis API without invasive modifications to the engine, as we leverage the same replication strategy. The internal AWS transaction log service provides strong consistency, durability across multiple AZs, and low latency. Writes to the log are only acknowledged once durably committed to multiple AZs, providing 11 9s of durability.

Relying on a loosely coupled transaction log service to provide durability allows it to scale independently from the in-memory engine. Therefore, the amount (and thus cost) of availability can be varied independently of the cost of durability. The cost of a MemoryDB node is dominated by DRAM which is sized relative to the working-set of the database. A transaction log is sized relative to write bandwidth and typically costs a small fraction of a MemoryDB node. Many MemoryDB customers operate shards with either a primary only or a primary with a single replica, but still receive durability across three AZs, which would not be possible if compute and storage were coupled.

Each shard in MemoryDB uses passive replication, where a primary replicates the mutative commands it executes to its transaction log. Specifically, MemoryDB intercepts the Redis replication stream, chunks it into records, and sends each record to the transaction log. The replicas read the replication stream sequentially from the transaction log and stream it into Redis. As a result, every replica holds an eventually consistent copy of the data set.

3.2 Maintaining Consistency

Redis is single-threaded and sequentially executes all commands it receives; however, it may lose committed writes across failovers due to its asynchronous propagation. MemoryDB provides linearizability by making propagation to the multi-AZ transaction log synchronous. Specifically, the conversion to synchronous replication faced the classic choice between write-ahead and write-behind logging [16]. We selected write-behind logging because this aligns with Redis replication model which generates replication information at the end of an operation. Write-behind logging allows MemoryDB to support non-deterministic commands, such as SPOP which removes a random element from a set, by replicating the effects of the command instead of the original command.

Due to our choice of using passive replication, mutations are executed on a primary node before being committed into the transaction log. If a commit fails, for example due to network isolation, the change must not be acknowledged and must not become visible. Other database engines use isolation mechanisms like Multi-Version Concurrency Control (MVCC) to achieve this, but Redis data structures do not support this functionality, and it cannot be readily decoupled from the database engine itself. Instead, MemoryDB adds a layer of client blocking. After a client sends a mutation, the reply from the mutation operation is stored in a tracker until the transaction log acknowledges persistence and only then sent to the client. Meanwhile, the Redis workloop can process other operations. Non-mutating operations can be executed immediately but must consult the tracker to determine if their results must also be delayed until a particular log write completes. Hazards are detected at the key level. If the value or data-structure in a key has been modified by an operation which is not yet persisted, the responses to read operations on that key are delayed until all data in that response is persisted. Replica nodes do not require blocking as mutations are only visible once committed to three AZs.

Data access on MemoryDB primary nodes is strongly consistent. Clients can opt-in reading from replicas by issuing Redis READONLY command. Each replica consumes updates from the transaction log, therefore providing a consistent point-in-time view of the data set. Reading from a single replica yields sequential consistency. Reading from multiple nodes, e.g., load balancing reads across multiple replicas, yields an eventually consistent view of the data set.

4 Availability, Recovery and Resilience

4.1 Leader Election

Redis cluster architecture is in the category of leader-follower with a majority-based quorum. It leverages a gossip protocol called cluster bus. Primaries from each shard constantly heartbeat each other via cluster bus. When a majority has not received heartbeats from a given primary, that primary is declared failed, and the majority will vote to elect one of the replicas of the failed primary. Replicas are chosen using a ranking algorithm, trying to promote the most-up-to-date replica based on the local perspective of each voting node. There is no guarantee that the elected replica observed all committed updates from the failed primary as Redis does not use consensus when accessing or updating data. For instance, if a primary node is isolated from the rest of the cluster, it continues servicing data, until a certain timeout, while on the healthy partition a replica could get promoted. In essence, Redis fails to satisfy some of the safety properties of a quorum-based replicated system: 1) leader singularity: at most there needs to be a single leader operating at any given point in time; 2) consistent failover: only a consistent replica can campaign and win leadership.

MemoryDB implements a leader election mechanism that always maintains leader election safety properties. We leverage the transaction log to build the leader election. MemoryDB ensures that only fully caught up replicas become eligible for promotion to primaries, therefore maintaining strong consistency across failures. Moreover, MemoryDB always ensures leader singularity, by leveraging a lease system that demotes failed primaries. Finally, MemoryDB does not require any cluster quorum for liveness, therefore improving availability over Redis cluster bus leadership mechanism.

4.1.1 *Building atop the Transaction Log*

The transaction log service provides a conditional *append API*. Each log entry is assigned a unique identifier, and it is required that each append request must specify the identifier of the entry it intends to follow as a pre-condition. Acquiring leadership is done by appending a specific log entry to the transaction log. Leadership is granted for a pre-determined lease [26].

4.1.2 *Consistent Failover*

Ensuring consistent failover becomes simpler leveraging the append API. Only replicas that have observed the latest write's unique identifier will succeed at appending the leadership entry to the log. When a replica is fully caught up with the transaction log, it is notified via a control message. When multiple replicas contend for leadership, only one will succeed and it will invalidate the pre-condition of any other concurrent append requests. This also yields an interesting property where old replicas, rejoining the cluster after failures, are naturally fenced and cannot contend for leadership.

MemoryDB leader election bypasses Redis cluster bus. It does not require a majority or a minimum number of nodes to run. Each replica only interacts with the transaction log service but not with each other. Only when a new primary is elected, the role change is propagated asynchronously via the cluster bus to inform the rest of the nodes in the cluster. The rest of the nodes can use this information to inform clients about role changes for minimal downtime.

4.1.3 *Leader Singularity*

We opted for the lease approach to provide in-memory performance in MemoryDB while maintaining consistency. Many consensus-based systems improve performance by using some form of leases, in which a node can satisfy read requests locally without having to commit an operation using the relatively more expensive consensus protocol. This optimization improves read throughput and latency, but also improves write performance by reducing the total number of operations that must be handled through consensus [30].

The lease granularity is at the Shard level. Leader and replica nodes cooperate to ensure leases remain always disjoint. Leaders periodically renew their lease by appending a lease renewal entry to the transaction log. Replicas observe the transaction log entries and start a pre-determined timer after observing a lease renewal. The time duration, called backoff, is ensured to be strictly greater than the lease duration. Replicas will refrain from campaigning for leadership during the backoff duration.

A primary that cannot renew its lease voluntarily stops servicing reads and writes at the end of its lease. When replica nodes do not observe any lease renewal entry in the transaction log after the backoff duration, they resume attempting campaigning for leadership.

In summary, building leader election on top of the transaction log service yields the following benefits. First, it improves liveness over Redis cluster bus leader election. It depends only on the availability of the transaction log service, an existing availability dependency regardless, instead of the additional availability of a majority in the cluster. Second, it strengthens consistency by strictly ensuring a single primary throughout failures including split-brain scenarios. If a primary cannot keep its lease, it self-demotes to prevent servicing stale data. A replica cannot campaign for leadership unless it observed all the updates in the transaction log. Finally, we leverage the *append* API offered by transaction log and its battle-tested consistency property that has been used by other Amazon production transactional systems. This simplifies the overall design and maintenance of the MemoryDB.

4.2 Recovery

In steady state, a primary periodically commits heartbeat messages in the transaction log to both indicate its liveness and extend its lease. When replicas have not observed any heartbeat after a timeout, they suspect the primary has failed, triggering a leader election to recover availability.

MemoryDB monitoring service constantly polls all MemoryDB replicas to monitor their health. This service is external to the data nodes and its polling results form an external view of cluster connectivity and health. Additionally, nodes within the same cluster constantly gossip with each other to form an internal view of cluster connectivity and health. When deciding a failure, both external view and internal view are consulted to improve failure detection accuracy. Once a node is determined to have failed, the monitoring service takes action to recover it. Depending on the failure mode, the database process could be restarted in-place or the underlying hardware could be replaced. New nodes always start up as replicas.

4.2.1 *Data Restoration*

MemoryDB is a strongly consistent database. After a failure, restoring previously committed data is on

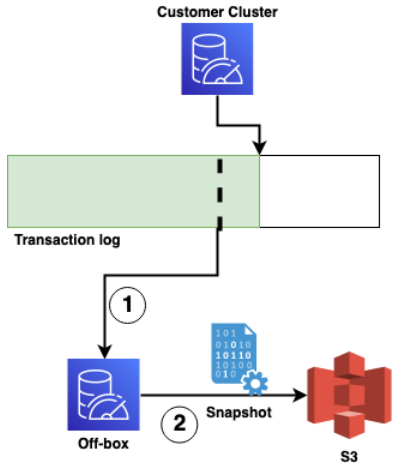


Figure 2: Off-box snapshot creation in MemoryDB. Off-box clusters are scheduled periodically and created using the latest generated snapshot. After that (1) the off-box cluster reads the transaction log up to the latest known update (recorded at the creation time of the off-box cluster). Finally, the off-box cluster takes a snapshot (2) and uploads it to S3, effectively making it the latest generated snapshot for this cluster.

the critical path to recovering availability. The efficiency of data restoration is critical to the mean-time-to-recovery (MTTR) of a cold restart.

We leverage Redis existing and battle-tested data synchronization APIs. First, a recovering replica loads a recent point-in-time snapshot and then replays subsequent transactions. While Redis requires the presence of a primary node to restore previously stored data, MemoryDB periodically creates snapshots and stores them durably in Simple Storage Service (S3) [2]. This allows MemoryDB to recover committed data to a node without the presence of a primary node. Recovering replicas fetch and load the latest snapshot from S3 and then replay from the transaction log. As result, data restoration becomes a process local to the restoring replicas where they do not interact with available peers in any way. Furthermore, this process allows recovery of multiple replicas to proceed in parallel without any centralized scaling bottleneck. S3 and the transaction log are separately scaled to potentially allow all replicas to restore data at the same time. It also avoids compounding node failures by not incurring extra workload on the healthy peers.

4.2.2 Off-box Snapshotting Redis creates snapshots of in-memory data by forking the database process. Leveraging the copy-on-write virtual memory management technique offered by the underlying operating system, the child process captures a point-in-time of the data set and serializes it into a snapshot file, while the original main process continues to accept mutations. This operation amplifies overall memory usage of the database and is CPU intensive. Some Redis users are accustomed to reserve extra memory space in an effort to offset this impact.

To improve this process, MemoryDB built *off-box* snapshot creation. Off-box clusters are ephemeral clusters that are invisible to customers and do not directly interact with customer clusters in any way. They share the same durable data sources (S3 and the transaction log) with customer clusters so that they can create snapshots on customer's behalf. S3 and the transaction log are scaled to accommodate the extra read workload from off-box clusters.

Off-box clusters essentially consist of shadow replicas of the customer clusters. Such off-box replicas are bootstrapped using the same data restoration procedure used by any recovering customer replica. Specifically, (1) each off-box replica restores a shard's latest generated snapshot in S3 and replays the transaction log up to a tail position recorded at the creation time of the off-box cluster and then stops. This re-creates a static data view that reflects a recent state on the customer cluster and is guaranteed to be fresher than any previous snapshot. Then, (2) each off-box replica dumps its data view into a new snapshot and uploads it to S3. By design, off-box replicas are not part of the customer cluster, therefore they are not subject to customer traffic and create snapshots by fully utilizing their available CPU and memory resources without interference.

While snapshotting could also be performed on customer replicas to optimize for cost, we decided not to for a couple of reasons: a) customers can still read from replicas and snapshotting is a compute and IO heavy operation that could impact client latencies that read from replicas; b) replicas play a crucial role in write availability in case a primary node fails, therefore if a replica is performing a snapshotting operation it could delay recovery, which can impact customer availability.

4.2.3 Snapshot Creation Scheduling A snapshot is usually a more efficient format than the transaction log for storing and restoring data. A snapshot captures each piece of data exactly once with its latest content in a compact form, whereas the transaction log contains all historical changes to a piece of data, many of which may have become irrelevant to its latest content. Therefore, to optimize data restoration efficiency and cold restart time, MemoryDB aims to bound the amount of transaction log to replay and make data restoration always snapshot-dominant.

Creating snapshots can be computationally expensive. MemoryDB's scheduling of snapshot creations strives to strike a balance between their freshness and cost. Specifically, MemoryDB constantly monitors the freshness of the latest snapshot of each MemoryDB cluster. Freshness of a snapshot can be visualized as its distance from the current tail of the transaction log. The fresher a snapshot, the less time a recovering replica would need to spend replaying the transaction log, the more snapshot-dominant and efficient a potential data restoration would be. How fast a snapshot freshness deteriorates is a function of both customer write throughput and customer data set size. Higher write throughput grows a snapshot "distance" from tail of the transaction log faster. Larger data set size makes creating a new snapshot take longer and indirectly allows the transaction log to grow more. MemoryDB monitoring service continuously samples this data on live clusters, calculates snapshot freshness based on these factors, and schedules new snapshot creations whenever the freshness is too stale.

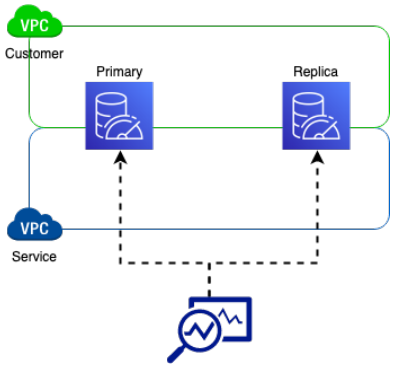


Figure 3: MemoryDB Monitoring. MemoryDB customer clusters are accessed via a customer-provided VPC. MemoryDB service uses another VPC to interact with customer clusters. A multi-tenant fleet monitors and manages MemoryDB clusters.

5 Management Operations

5.1 Cluster Management

MemoryDB system is successful at scale as a result of the management capabilities provided by the service (henceforth known as the control plane). The control plane is responsible for processing customer provisioning requests and performing cluster updates and upgrades, including coordinating scaling activities. The control plane is also responsible for maintaining high availability for MemoryDB by quickly diagnosing and remediating failures at a cluster-level. Recovery is coordinated by the control plane.

The control plane is a regional multi-tenant service which manages a fleet of single-tenant clusters on behalf of customers. Each request to create a new MemoryDB cluster provisions the specified number of Amazon EC2 [7] instances and the required number of Multi-AZ transaction logs, and then configures the nodes into the requested topology. The control plane uses the appropriate AWS Key Management Service (KMS) keys (either customer-owned or service) [13] in an envelope encryption strategy (i.e., plain-text data with a data key, and then encrypting the data key under another key [13]) to encrypt data at rest on the MemoryDB nodes themselves and on the multi-AZ transaction log.

During the cluster creation request, the customer provides a Virtual Private Cloud (VPC) [3]. When a customer creates shards with at least one replica, the nodes are placed in different AZs to ensure no downtime in the event of a single AZ failure. The control plane is responsible for attaching the cluster's nodes into a customer VPC, provisioning stable DNS endpoints to point to the nodes, vending TLS certificates if needed, as well as pushing configuration such as Access Control Lists (ACLs) to each node. These activities are coordinated at a cluster-wide basis and parallelized as appropriate across shards and/or nodes.

The control plane coordinates operations like patching and scaling. Instead of a traditional blue/green [4] deployment strategy, MemoryDB uses a rolling N+1 upgrade process: instead of upgrading nodes in-place, new nodes running with the new software are provisioned. This mitigates the impact on cluster availability by allowing all nodes to serve traffic while an upgrade is occurring. Similarly, the process of scaling out a cluster involves adding a new shard composed of new nodes and gradually moving Redis slots from existing shards to new shards. This process is orchestrated centrally and is discussed in more details in section 5.2

The monitoring service fetches data from all nodes in a cluster every 5 seconds to understand cluster health. It serves as a watchdog for cluster configurations, fixing those that are valid (such as detected dead replicas) and alarming on those that are invalid (such as only replicas detected in a shard).

5.2 Scaling

The size of a MemoryDB cluster is measured in three dimensions: number of shards, number of replicas per shard and EC2 instance type (CPU & memory per instance). Control plane APIs are provided to dynamically adjust any one of these three dimensions on a running cluster without significant interruption. The APIs can be invoked manually or programmatically.

Scaling the number of replicas is the simplest operation. To decrease the replica count, one replica from each shard is selected and terminated, releasing the associated EC2 instance. To increase the replica count, a new EC2 instance for each shard is created and provisioned. Once the new EC2 instance is operational it reloads the most recent snapshot for the shard from S3 and then replays its transaction log. Once the tip of the transaction log is reached the replica joins the cluster, advertising its availability via the Redis cluster bus.

Scaling the instance type is performed as an N+1 rolling update. Replicas of the new instance type are created using the procedure described above. Once the new replica has joined, the control plane selects a node of the previous instance type (replicas first, primary last) to decommission, causing a leader election in the case of the shard primary. A collaborative leadership transfer, where the old instance actively hands over leadership, minimizes downtime. In the case where the new instance type is smaller than the old instance type, it's possible to run out of memory, in which case the scaling operation is reverted, restoring the original instance type.

Scaling the number of shards requires transferring one or more slots between shards and either creating shards at the beginning of the operation (scaling out) or destroying shards at the end of the scaling operation (scaling in). The shard creation/destruction involves provisioning/terminating nodes as in the replica scaling operations described above plus a per-shard transaction log creation/destruction. The slot transfer is divided into two phases: data movement and slot ownership transfer. The data movement phase is conceptually similar to a Redis replica synchronization, but limited to a specific slot, such that keys for the slot being transferred must be serialized and transmitted (from the source primary node to the target primary node) while continuing to allow operations that may mutate those same keys. As a result, the transferred data includes both serialized keys and replication stream mutations of keys already transmitted. The target primary commits all messages to the transaction log, allowing its replicas to reach the same state for the slot.

Before the ownership transfer can be initiated, the source primary ensures that all data has been transferred by blocking all

new incoming write operations for the slot and waiting for any in-progress write operations to complete execution and propagation to the source and target transaction logs. At this point, a data integrity handshake with the target is performed to validate the correct transfer of data — any error up to this point (out of memory, network error, validation failure, etc.) is easily recovered from by simply abandoning the transfer operation, i.e., resuming write operations and directing the target to delete all the transferred data.

In Redis, slot ownership is controlled and communicated through the eventually consistent cluster bus. This mechanism is known to have several failure modes, resulting in corruption or loss of stored data. Consistent with our principles of minimizing divergence with the open-source code base, communication of slot ownership remains a cluster bus responsibility. However, slot ownership is stored in the transaction log and changes to slot ownership are performed using a 2 Phase Commit (PC) protocol of durably committed messages between the old and new owner of a slot. Once the slot ownership has been transferred, the new owner begins accepting writes while the old owner of the slot properly responds with a redirect for operations on the moved slot and starts deletion of all the transferred data in a rate-limited background task. Typically, the duration of the write unavailability for the slot during the ownership transfer phase is limited to a few network round trips and the transaction log update latencies. Failures at the source or target, for example due to lease expiration 4.1, can be recovered as the progress of the 2PC is recorded in the transaction log. After a primary node failure (source or target) recovery, the ownership transfer protocol can continue.

6 Evaluation

The goal of this section is to evaluate the cost of durability in MemoryDB. There are two components to durability in MemoryDB: (1) steady state writes that are committed to the transaction log; (2) periodic snapshots uploaded to S3. To evaluate the performance in steady state, we use a benchmark to illustrate the performance profile of MemoryDB under different types of workloads. We use OSS Redis as a baseline to compare to MemoryDB. After that, we focus on the snapshotting component by comparing the Redis OSS snapshotting facility performance to MemoryDB purpose-built offbox.

6.1 Performance Benchmark

6.1.1 Setup We evaluated the performance of all (graviton3) supported instance types in MemoryDB from r7g.large up to r7g.16xlarge. Both MemoryDB and OSS Redis use engine version 7.0.7. We tested 3 types of workloads: Read Only, Write Only, and Read Write Mixed workloads. In Read Only workload, each clients send a GET request to Redis server back to back (without pipelining). In Write Only workload, SET command is used. In Read Write Mix workload, 80% of the requests are GET and 20% of the requests are SET. We use 10 EC2 instances each running a redis-benchmark process to drive traffic to MemoryDB and Redis. The 10 EC2 instances are launched in the same AZ as the MemoryDB and Redis to minimize network latency. Before the test, nodes are pre-filled with 1 million keys, so that GET request will have 100% hit rate. We configure each redis-benchmark process with 100 client connections and 100 byte values. We use simple GET/SET operations to get a consistent performance baseline as opposed to using more compute heavy operations that manipulate (potentially large) data structures. Redis supports threaded IO which offloads IO operations to background Threads. MemoryDB supports Enhanced IO [8], a similar internal feature allowing the engine to offload IO to background threads. MemoryDB Enhanced IO has more advanced features, for example, it can multiplex clients into a single connection therefore reducing the overhead IO fan-in/fan-out. We configure Redis with the same number of IO threads as MemoryDB for each instance type. Since Redis does not support SSL with IO threads, we disable TLS encryption and authentication.

6.1.2 Benchmark Result

6.1.2.1 Throughput Evaluation Figure 4 illustrates the maximum throughput observed on different instance types for read-only and write-only workloads. We observe that for read-only workloads (Figure 4a), both Redis and MemoryDB achieve comparable throughput for instance types lower than 2xlarge, up to 200K Op/s. Starting with 2xlarge, MemoryDB outperforms Redis achieving 500K Op/s across all instance types, while Redis achieves a maximum of 330K Op/s.

This illustrates that MemoryDB performs well on read-only workloads. MemoryDB outperforms Redis as MemoryDB Enhanced IO Multiplexing [8] aggregates multiple client connections into a single connection to the engine, improving processing efficiency and delivering higher throughput.

With write-only workloads (Figure 4b), we can see that Redis outperforms MemoryDB on all instance types, achieving a maximum throughput close to 300K Op/s, whereas MemoryDB achieves up to 185K Op/s.

MemoryDB commits every single write to the multi-AZ transaction log, resulting in higher request latency. Given the same workload in terms of number of clients making sequential blocking requests, therefore, MemoryDB delivers lower throughput on write-only workloads compared to Redis. With workloads with higher number of clients, pipelining, or large payload sizes, our experiments show that a single shard can achieve up to 100MB/s write throughput in MemoryDB.

6.1.2.2 Latency Evaluation Figure 5 shows the latency of Redis and MemoryDB when varying the offered throughput, on an r7g.16xlarge instance type, for different workloads.

We observe that for read-only workloads (Figure 5a), Redis and MemoryDB have a similar latency profile both on median (sub-millisecond) and tail latencies (less than 2 milliseconds on p99). For write-only (Figure 5b) workloads, Redis delivers sub-millisecond median latencies and up to 3 milliseconds on p99, while MemoryDB delivers 3 milliseconds on median and up to 6 milliseconds on p99. For mixed read-write workloads (Figure 5c), both Redis and MemoryDB offer sub-millisecond median latencies, while Redis delivers up to 2 milliseconds, and MemoryDB up to 4 milliseconds p99 latencies.

This shows that MemoryDB offers sub-millisecond median latencies for read and mixed read-write workloads and single-digit millisecond latencies for write and tail mixed read-write workloads, while ensuring multi-AZ durability for every single write.

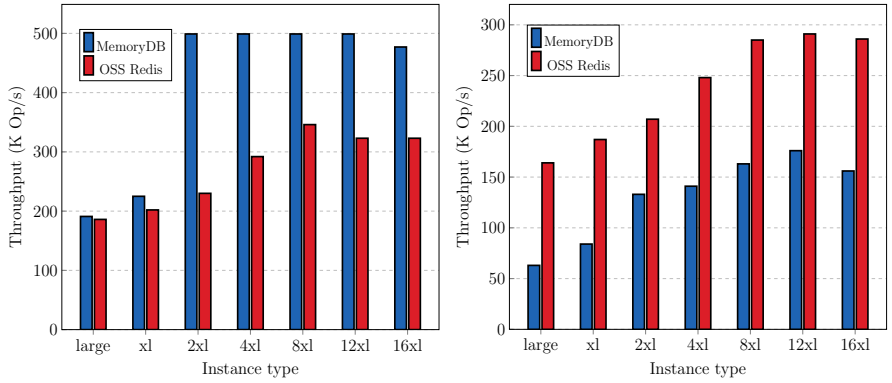


Figure 4: MemoryDB and Redis throughput for read-only (a) and write-only (b) workloads.

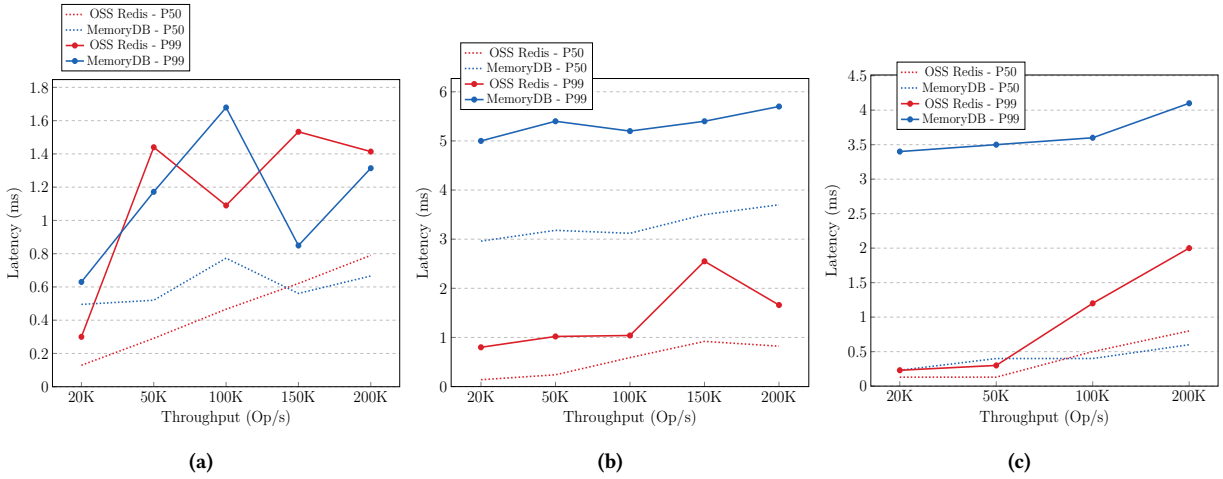


Figure 5: MemoryDB and Redis latencies when varying the offered throughput for read-only (a) write-only (b) and mixed read-write (c) workloads, on 16xlarge instance type.

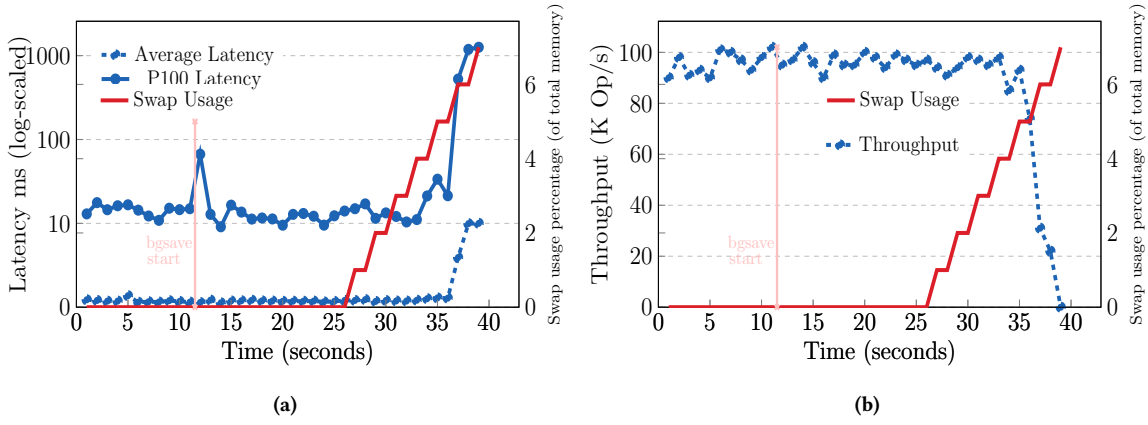
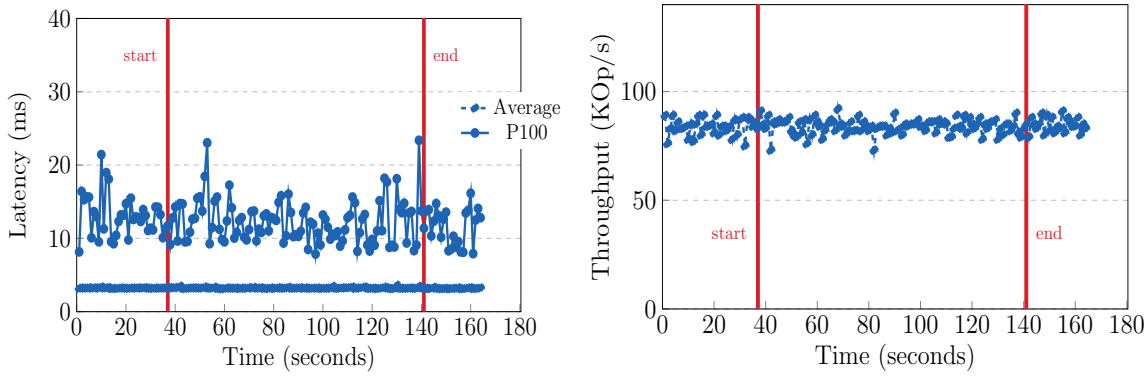


Figure 6: Client perceived latency (a) and throughput (b) during BGsave in a memory constrained setup with Redis.



(a) (b)

Figure 7: Client perceived latency (a) and throughput (b) during BGsave when off-box technique is used.

6.2 Snapshotting Evaluation

Redis uses Background Save (BGSave) process to perform snapshots. BGSave forks the Redis process to create a child process to perform snapshots. This child process creates a point in time snapshot of the database by iterating over the entire keyspace to serialize data to disk. During this serialization process, a Copy on Write (COW) can occur if a memory page is modified by Redis (parent process), resulting in the memory page being copied so that the child process' corresponding memory page remains intact. COW can cause excessive memory to be accumulated when there is a heavy write workload. In the worst case, it could double the memory consumption which could lead to high swap usage, causing significant latency increase and throughput degradation.

The remainder of this section answers the following questions:

- What is the overhead of BGsave in Redis?
- What is the overhead of snapshotting in MemoryDB using off-box?

To setup the experiment we use an instance with 2 vCPU and 16GB RAM. The max memory is configured as 12GB. The data has been pre-filled with 20 million keys, each key value pair is 500 Bytes. We use larger payload sizes compared to section 6.1.2 to increase memory pressure faster. 100 clients are issuing GET commands to measure throughput and latency, while another 20 clients are issuing SET commands.

When the snapshot process is running, we record the average throughput as well as the average latency and p100 latency. We explicitly choose p100 to display tail latency as the number of latency samples in a single second is lower compared to the entire run of the experiment 6.1.2.

6.2.1 What is the overhead of BGsave? Figure 6 presents an evaluation of the latency and throughput impact of COW on Redis. We observe that when BGsave starts there is no impact on throughput. However, there is a spike on P100 latency reaching up to 67 milliseconds for request response times. This is due to the fork system call which clones the entire memory page table. Based on our internal measurement, this process takes about 12ms per GB of memory.

Once the instance exhausts all the DRAM capacity and starts to use swap to page out memory pages, the latency increases and the throughput drops significantly. This is because CPU is stalled while waiting for spilling memory pages to disk before it can continue performing COW. The tail latency increases over a second and throughput drops close to 0 as swap goes beyond 8% of total memory, which is effectively an availability outage from a client's perspective.

To prevent this in practise, users of Redis would need to reduce the database available memory to at most half of the host available DRAM to prevent write workloads from driving the system to swap and causing an availability impact, or run snapshotting during off-peak hours when little to no write traffic is expected.

6.2.2 What is the overhead of snapshotting in MemoryDB using off-box? In MemoryDB, customers can use a variety of instances types, including instances with memory capacity as low as 1.37GB and 2 vCPUs. In order to provide durability with snapshots while ensuring performance, MemoryDB performs the snapshot on off-box clusters. Figure 7 depicts the throughput and latency on a MemoryDB cluster, while an off-box cluster snapshot process is running in parallel. We observe average latencies hovering around 1 millisecond, while the maximum latency varies between 10 milliseconds and 20 milliseconds. The p100 is higher compared to the number reported in section 6.1.2 because we are running a mixed read/write workload while snapshotting, values are 5x larger, and tail read latency would be impacted by the commit latency (i.e., if a read is trying to access an uncommitted key).

When snapshotting starts, we observe that the throughput and latencies are stable before and do not change throughout the process, as well as after it ends. Given the off-box process spins up a cluster isolated from the customer cluster, there is no impact on the customer workload during snapshotting in MemoryDB. As a result, MemoryDB customers need not to reserve any memory capacity for snapshotting nor worry about coordinating snapshotting during off-peak hours.

7 Validating and Maintaining Consistency at Scale

7.1 Consistency During Upgrades

MemoryDB maintains version currency with open-source Redis. As a result of our decoupling of in-memory storage and durability, we can leverage Redis as an in-memory execution engine. We are able to merge open-source changes into MemoryDB without much difficulty, provided they do not fundamentally alter the replication strategy. MemoryDB allows customers to elect to upgrade engine versions to gain additional functionality such as new data structures or commands.

As described earlier, we use a N+1 rolling upgrade strategy to maintain availability during upgrades. During this upgrade process, replicas are upgraded first and leader node is upgraded at the very end to preserve read throughput capacity. To maintain availability, we cannot force all nodes to upgrade at the same time in a transactional fashion. Therefore, clusters undergoing an upgrade operation could have mixed versions during a transient time period. This can cause inconsistencies. For example, a leader node running a newer engine can send a newly introduced command to the transaction log, while replicas running older engines observe those command, in the worst-case misinterpreting those commands. If the leader with the new engine fails and the replicas with the older engine becomes leader, then this could lead to inconsistency.

To cope with this problem, we have developed an upgrade protection mechanism. We protect the replication stream by indicating which engine version produced it. If a replica with an older engine version observes a replication stream originating from a newer version than the one it is currently running, it stops consuming the transaction log. To ensure that the cluster remains available, even during failures while upgrading, the control plane coordinates off-box processes such that we take a snapshot with the oldest engine version running in the cluster. This allows nodes that are still running older engine versions to be replaced in case of failures during the upgrade process.

7.2 Verifying Correctness

At AWS we strive to build services that are simple for customers to use. That external simplicity is built on a hidden substrate of complex distributed systems, and MemoryDB is no exception. High complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend [31].

7.2.1 Snapshot Correctness Verification MemoryDB verifies every newly created snapshot in production to make sure its consistency invariant holds: that snapshots are equivalent to their corresponding prefix of the transaction log. Specifically, MemoryDB maintains a running checksum of the entire transaction log and periodically injects the current checksum value into the transaction log itself. A snapshot stores the checksum value as of the transaction log prefix it captures as well as a positional identifier for the last log entry in the prefix. A snapshot also stores a checksum covering the data it contains. MemoryDB verifies the correctness of a snapshot by rehearsing restoring it on an off-box cluster. First, it validates the contents of the snapshot itself with the checksum covering the data stored. Then, it uses its stored positional identifier to locate the subsequent transaction log to replay. While it is replaying the transaction log, it uses the snapshot checksum as a basis to recalculate a running checksum and compare it against the checksum persisted in the transaction log. The verification would fail if the snapshot's checksum does not match the transaction log prefix it captures. Only successfully verified snapshots will be made available to customers.

7.2.2 Consistency Validating the functional correctness of Redis API while maintaining strong consistency is not trivial. Similarly to [19], we decompose the system correctness validation which allows applying a diverse suite of formal methods tools to best check each property.

7.2.2.1 Formal Verification MemoryDB durability dependencies use various tools including formal verification to ensure their correctness. S3 uses TLA+ and lightweight formal methods [28, 31] to model and test various components. The internal transaction log replication protocol is modelled and verified using TLA+. MemoryDB uses P [23] as well for new feature development, which proved helpful in reasoning about the overall system and in catching bugs early in the development process.

7.2.2.2 Consistency Testing Framework Applying formal approaches to test the Redis API proved to be challenging, as its implementation is fast moving and frequently changing. We needed to ensure the results of validation remain relevant regardless of the Redis implementation, while validating properties of APIs and consistency under failure modes. We use porcupine [17], a linearizability checker, to test MemoryDB consistency. Porcupine takes as input a concurrent history of client commands, and outputs whether that history is linearizable. To ensure the framework has full coverage over Redis API, we parse the API specification provided by the engine and generate commands based on the output. Similarly to [19], we leverage argument biasing to improve our testing coverage, especially around edge-cases. Overall, this helped us improve our confidence testing MemoryDB durability components and its dependencies.

8 Related Work

8.1 Disaggregated databases

A number of distributed database systems [14] have been built for disaggregated storage [20, 35]. Amazon Aurora [35] offloads redo processing to a multi-tenant, scale-out storage service. Sinfonia [14] and Hyder [18] are systems that abstract transactional access methods over a scale out service, allowing database systems to be implemented using these abstractions. PolarDB [20] uses Remote Direct Memory Access (RDMA) to connect disaggregated storage with computation nodes. MemoryDB leverages database nodes for storage and command processing while offloading replication and durability to a scale-out transactional log service.

8.2 Log-based Replication

Log-based replication has been extensively used by consensus algorithms [29, 33] as well as by distributed storage systems [24, 35, 36]

as a way to provide durability. Multi-paxos [29] and Raft [33] are well-known protocols used to build consistent replicated logs. MemoryDB implements a similar protocol to perform consistent log-based replication, with similar safety properties: a single node can become leader at a given point in time; a committed log entry will always appear in any future leader state; a node can only become leader if it observed all committed log entries. MemoryDB improves liveness by leveraging a scale-out transaction log service to build consensus and durability.

8.3 In-memory databases

In recent years, large-scale Web applications systems have found DRAM indispensable to meet their performance goals which led to the emergence of a number of in-memory NoSQL storage systems [6, 12, 32, 34] and architectures such as anti-caching [22]. A number of relational databases use memory as their main storage medium to improve performance [25, 27]. Redis emerged as one of the most popular in-memory storage systems given its rich data model. However, it is hard for users to rely on Redis as their primary database due to its weak durability guarantees. MemoryDB is a cloud native memory-based database providing strong consistency, 11 9s durability, and 4 9s availability.

9 Conclusion

This paper presents Amazon MemoryDB, a fast and durable in-memory storage cloud-based service. A core design behind MemoryDB is to decouple durability from the in-memory execution engine by leveraging an internal AWS transaction log service. In doing so, MemoryDB is able to separate consistency and durability concerns away from the engine allowing to independently scale performance and availability. To achieve that, a key challenge was ensuring strong consistency across all failure modes while maintaining the performance and full compatibility with Redis. MemoryDB solves this by intercepting the Redis replication stream, redirecting it to the transaction log, and converting it into synchronous replication. MemoryDB built a leadership mechanism atop the transaction log which enforces strong consistency. MemoryDB unlocks new capabilities for customers that do not want to trade consistency or performance while using Redis API, one of the most popular data stores of the past decade.

References

[1] [n. d.]. Amazon elasticache – managed caching service – Amazon Web Services. https://aws.amazon.com/elasticache/.

[2] [n. d.]. Amazon S3. https://aws.amazon.com/s3/.

[3] [n. d.]. Amazon Virtual Private Cloud (Amazon VPC). https://aws.amazon.com/vpc/.

[4] [n. d.]. Blue/Green Deployments on AWS. https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/bluegreen-deployments.html.

[5] [n. d.]. DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed on 01/04/2023.

[6] [n. d.]. Hazelcast. https://hazelcast.com/.

[7] [n. d.]. AWS EC2. https://aws.amazon.com/ec2/.

[8] [n. d.]. MemoryDB Features. https://aws.amazon.com/memorydb/features/.

[9] [n. d.]. Optimize Redis Client Performance for Amazon ElastiCache and MemoryDB. https://aws.amazon.com/blogs/database/optimize-redis-client-performance-for-amazon-elasticache/.

[10] [n. d.]. Redis cluster specification. https://redis.io/docs/reference/cluster-spec/.

[11] [n. d.]. Redis Enterprise. https://redis.com.

[12] [n. d.]. Redis in-memory data store. https://redis.io.

[13] [n. d.]. AWS Key Management Service (AWS KMS). https://aws.amazon.com/kms/.

[14] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. *SIGOPS Oper. Syst. Rev.* 41, 6 (oct 2007), 159–174. https://doi.org/10.1145/1323293.1294278

[15] Aditya Akella, Amin Vahdat, Arjun Singhvi, Behnam Montazeri, Dan Gibson, Hassan Wassel, Joel Scherpelz, Milo M. K. Martin, Monica C Wong-Chan, Moray Mclaren, Prashant Chandra, Rob Cauble, Sean Clark, Simon Sabato, and Thomas F. Wenisch. 2020. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. New York, NY, USA, 708–721.

[16] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. *Proc. VLDB Endow.* 10, 4 (nov 2016), 337–348. https://doi.org/10.14778/3025111.3025116

[17] Anish Athalye. 2017. Porcupine: A fast linearizability checker in Go. https://github.com/anishathalye/porcupine.

[18] Phil Bernstein, Colin Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR* (cidr ed.). 9–20. https://www.microsoft.com/en-us/research/publication/hyder-a-transactional-record-manager-for-shared-flash/ Best Paper Award.

[19] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

[20] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 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)*. USENIX Association, Santa Clara, CA, 29–41. https://www.usenix.org/conference/fast20/presentation/cao-wei

[21] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18), Houston, TX, USA* (2018 acm sigmod international conference on management of data (sigmod '18), houston, tx, usa ed.). ACM. https://www.microsoft.com/en-us/research/publication/faster-concurrent-key-value-store-place-updates/

[22] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1942–1953. https://doi.org/10.14778/2556549.2556575

[23] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-Driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 321–332. https://doi.org/10.1145/2491956.2462184

[24] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 1037–1048. https://www.usenix.org/conference/atc22/presentation/elhemali

[25] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.* 40, 4 (jan 2012), 45–51. https://doi.org/10.1145/2094114.2094126

[26] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. *SIGOPS Oper. Syst. Rev.* 23, 5 (nov 1989), 202–210. https://doi.org/10.1145/74851.74870

[27] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (aug 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[28] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (may 1994), 872–923. https://doi.org/10.1145/177492.177726

[29] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

[30] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association

for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2671001

[31] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (mar 2015), 66–73. https://doi.org/10.1145/2699417

[32] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[33] 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.

[34] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (aug 2015), 55 pages. https://doi.org/10.1145/2806887

[35] 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 *SIGMOD 2017*. https://www.amazon.science/publications/amazon-aurora-design-considerations-for-high-throughput-cloud-native-relational-databases

[36] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a Replicated Logging System with Apache Kafka. *Proc. VLDB Endow.* 8, 12 (2015), 1654–1655. http://dblp.uni-trier.de/db/journals/pvldb/pvldb8.html#WangKSPZNRKS15

Received 30 November 2023; accepted 4 February 2024