Garnet: A Next-Generation Cache-Store for Accelerating Applications and Services

Badrish Chandramouli Vasileios Zois Ted Hart Tal Zaccai
Lukas M. Maas Yoganand Rajasekaran Darren Gehring
Microsoft Research
{badrishc,vazois,tedhar,talzacc,lumaas,yrajas,darrenge}@microsoft.com

ABSTRACT

Remote cache-stores have seen a dramatic rise in importance in recent years, fueled by a surge in data-driven applications. Most prior database research has focused on various aspects of traditional *key-value stores* with string values and a simple `get/set` based remote interface. However, modern cache-stores such as Redis offer a significantly richer interface that has witnessed unprecedented popularity and broad adoption across the developer community. The interface and use cases for such cache-stores in both end-user applications and large-scale services translate to new requirements on storage, scale, complex data type support, and durability.

Garnet is a new cache-store that adopts the Redis wire protocol for compatibility, but rethinks from a database perspective how such a modern cache-store system should be designed from the ground up to meet these requirements. Research insights across the storage, network, and cluster stack allow Garnet to support the large Redis interface as a drop-in replacement, yet achieve stronger database features—thread- and node-scalability, durability, transactions—and better end-to-end performance (up to 100× higher throughput and 4× lower latency at high percentiles). These results translate to lower end-to-end costs for real-word applications and services.

PVLDB Reference Format:
Badrish Chandramouli, Vasileios Zois, Ted Hart, Tal Zaccai, Lukas M. Maas, Yoganand Rajasekaran, and Darren Gehring. Garnet: A Next-Generation Cache-Store for Accelerating Applications and Services. PVLDB, 19(2): 224 - 237, 2025.
doi:10.14778/3773749.3773760

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/microsoft/garnet.

1 INTRODUCTION

Caches and storage systems have seen a dramatic rise in importance in recent years, fueled by a surge in data-driven applications and services. More than a decade of database research has gone into various aspects of traditional *key-value stores* [4–6, 30, 32, 39] that focus on string values with a `get/set` based interface.

However, *rich remote cache-stores* (or, cache-stores for brevity) have quickly become the standard in recent years. We define a cache-store as a storage server that exposes one or more remote endpoints that adhere to a well-defined *wire protocol* and a rich (or expressive) *command set*. A cache-store can be deployed in either *standalone* (single instance) or *cluster* (sharded and/or replicated set of instances) mode. Clients are typically remote, and may be implemented in any programming language—they interact with the cache-store using the specified protocol and command set.

Example cache-stores are Redis [44], its recent fork Valkey [60], KeyDB [25], and Dragonfly [12]—these systems offer a significantly richer interface that is now adopted across the developer community, with clients in nearly every language. All cloud vendors offer services around cache-stores, such as Google MemoryStore, Amazon MemoryDB and ElastiCache, and Microsoft's Azure Redis Cache. More broadly, the in-memory data store/cache market is projected to grow from $8 billion in 2024 to $30 billion by 2033 [22].

Beyond the `get/set` interface, cache-stores offer advanced value data types such as `SortedSet`, `List`, `Hash`, `Set`, and `Geo`, per-key expiration times, analytics capabilities (HyperLogLog [20] and bitmap sketches), transactions, and stored procedures. Cache-stores are used in a variety of ways, for example, as a scalable caching tier in front of databases, to decouple and share session state across application instances, and even directly as a fully durable database [47].

1.1 Requirements for Modern Cache-Stores

Our discussions with large application services at Microsoft indicate a common usage pattern. Applications usually start by directly using database services. As they grow in popularity and usage, however, the need to lower service costs and reduce latency results in the introduction of a cache-store layer in front of the database. The rich cache-store interface eventually leads to applications using them to store additional state beyond what the database holds, e.g., sorted sets with per-key expiration are a convenient way to implement a *sliding window rate limiter* [56] for a search service. Other scenarios include gaming leaderboards, recommendation systems, real-time streaming analytics, job queues, content and session state caching, distributed locks, counters, social media state, and LLM prompt caching [47, 50, 61]. This increased usage in turn leads to greater expectations in terms of data volumes served, performance targets, and durability guarantees for the cache-store. A survey found that over 70% of Redis users treated it as their database and not just a cache [47]. Based on these and observations, we identify the following unique requirements for modern cache-stores:

- *Larger-Than-Memory but Memory-Optimized (*`LTM-MO`*)*: Given the increasing cost of memory and scalable IOPS of modern storage, users want cache-stores over larger-than-memory data sizes. Yet, they need to be highly optimized for the common case where the stored data (or its working set) fits in memory.

Proceedings of the VLDB Endowment, Vol. 19, No. 2 ISSN 2150-8097.
doi:10.14778/3773749.3773760

Table 1: Comparing RESP cache-stores (normalized to Redis).

Feature	Redis	Valkey	KeyDB	Dragonfly	Garnet
Node Throughput (§ 8.3)	1×	1×	0.6×	10×	108×
Latency, batch size 16 (§ 8.4)	1×	1×	1.4×	0.67×	0.25×
Kernel Bypass Ability (§ 3)	✗	✓	✗	✗	✓
Tiered Storage (§ 4)	✗	✗	✓	✗	✓
Decoupled Storage (§ 4)	✗	✗	✗	✗	✓
Tunable Durability (§ 5)	✓	✓	✓	✗	✓
Checkpoint Speed (§ 5, § 8.6)	1×	1×	0.57×	59×	257×
Scale-out Speed (§ 6, § 8.7)	1×	1×	0.88×	11×	94×

- *Bare-Metal Performance*: Cache-stores are deployed to reduce latency and cost as compared to database servers, so high performance that is limited only by the underlying hardware is a key requirement. Concretely, users want (1) high throughput, saturating memory bandwidth for memory operations, disk IOPS for disk operations, or network bandwidth (remote cache-stores are typically limited by the latter); (2) low median and 99th percentile end-to-end latencies, with the ability to leverage kernel bypass networking [23] for lower latency; (3) scalability with increasing cores and nodes; and (4) fast non-blocking checkpoints.
- *Complex Types and Operations*: The cache-store should work efficiently with not just raw string values, but also complex data types represented as *heap objects*. We define a heap object as any in-memory data structure, often with its own internal pointers, such as a linked list for `List`. Users expect many data types with hundreds of commands over them (e.g., more than 250 in Redis). Given the large and evolving number of commands, a narrow and safe storage interface is necessary so that developers can add and maintain these types and operations without understanding the underlying subtle high-performance storage implementation.
- *Tunable Durability*: Given the diverse use cases and a user preference [57] to not maintain a separate durable database, a cache-store should be configurable with a range of durability options such as ephemeral, periodic checkpoints, lazy durability (continuously commit in the background), and full durability (commit before acknowledging writes). Users should also be able to configure a variable number of replicas and shards in a cluster mode without loss of performance.

1.2 Today's Cache-Store Architectures

Traditional key-value store research [9, 21] has focused on the `get`/`set` interface on strings, with sharded and replicated architectures. The Redis (and now Valkey [60]) cache started as a simple in-memory hash table, but its protocol—called the *Redis Serialization Protocol* (*RESP*)—and command set have evolved to meet developer needs over time (250+ commands to date). Complex data types are implemented as raw C logic and instantiated in main memory, making new data structure additions difficult and error-prone.

The simplest server architecture for a cache-store is a single-threaded storage component (e.g., a hash table) accessed by a multi-threaded network layer. Incoming requests can either be multiplexed into a single work queue or directly access the storage component under a lock. These two architectures correspond to Redis and KeyDB respectively, but achieve limited throughput on modern servers. A slightly improved design replaces the single work queue with one queue per server core, where each queue handles some partition of the key space (similar to Redis Enterprise [46]). While this design increases server parallelism, it suffers from the high cost of examining incoming packets, breaking batches by routing individual requests to the owning worker, and collating responses to be returned to clients in the correct order. It can also suffer from imbalance across queues with skewed workloads. Dragonfly is similar, based on shared-nothing shards [13] of in-memory Dash [35] hash tables. Alternatively, one could run multiple processes of the single-work-queue server as an "intra-node cluster" and rely on clients to route requests. However, this design leads to a quadratic increase in connections, fewer batching opportunities per connection, and client-side overhead. In Section 8.5, we evaluate these designs and show that they each have performance limitations. Features of the leading RESP cache-stores are compared in Table 1.

1.3 A New Design

This paper introduces Garnet[1], a new cache-store that adopts the RESP protocol and command set for compatibility with existing clients, but rethinks the cache-store design from the ground up. Garnet offers new research insights across the storage, network, and cluster stack that together allow the system to support a rich and ever-growing command set, while satisfying all the cache-store requirements outlined earlier. Garnet's research novelty stems from the confluence of four major design factors:

(1) A new shared-memory and tiered-device storage engine called Tsavorite[2] with a *narrow-waist* [8] storage interface of just *five basic operations*—the rich RESP command set sits as *user logic* atop this narrow interface. Tsavorite is backed by a latch-free thread-scalable `LTM-MO` storage design with a memory-reuse technique called *revivification*. Unlike traditional storage engines, Tsavorite handles string and heap object values as first-class components with a unique *dual store* design.
(2) A pluggable thread-switch-free network layer that allows both standard TCP and kernel bypass network stacks with TLS encryption, coupled with a fast protocol parsing layer. This shared-nothing layer interacts directly with Tsavorite by mapping hundreds of RESP commands to the five Tsavorite operations. In shared memory, cache coherence is used to bring the data to the network/query thread, maximally leveraging the high memory-to-CPU bandwidth of modern machines.
(3) Tunable durability, that lets Garnet operate in different modes, from a pure cache to a durable database. This is enabled by non-blocking checkpoints coupled with a *deterministic operation log* (*DOL*) abstraction. DOL sits atop tiered storage with a configurable commit depth. The fate and consistency of the dual stores are coupled by the DOL and checkpoint mechanisms.
(4) A full-fledged cluster design with fast migration and replication enabled by protocols based on an *epoch-protected state machine* (*EPSM*) abstraction. More generally, EPSMs are used throughout Garnet as a common non-blocking control plane mechanism.

Experiments show that Garnet can achieve up to 100× higher throughput and 4× lower latency end-to-end than comparable cache-stores. These results translate to lower costs for applications and services. Garnet is already widely deployed in large-scale scenarios across Microsoft, and its recent open-source release has

[1]Garnet stands for a gemstone known for its vibrant red hues, symbolizing vitality.
[2]Tsavorite is a rare form of green gemstone that is part of the Garnet family.

Table 2: Example commands in RESP.

Type	Command	Description
raw string	`GET key`	Retrieves the value for given key.
raw string	`SET key value`	Assigns a value for a key.
raw string	`SETNX key value`	Sets value if key does not already exist.
raw string	`SETXX key value`	Sets value if key already exists.
raw string	`SETEX key seconds value`	Sets value with an expiration time.
`List`	`LPUSH key value`	Adds value to the list's head.
`List`	`RPOPLPUSH src dest`	Moves element from source key (`src`)'s tail to destination key (`dst`)'s head.
`SortedSet`	`ZADD key score member`	Adds member with specified score to the sorted set stored at key.
`Set`	`SPOP key [count]`	Removes and returns one or more random members of the set stored at key.

seen strong user traction, with over 11K stars, more than 600 forks, and an active developer community on GitHub. Open-source developers regularly contribute new commands (more than 100 to date) to Garnet, because its layered storage design means they do not need to understand how Tsavorite works in order to contribute.

The rest of this paper is organized as follows. Section 2 covers background, while Section 3 introduces Garnet's layered architecture, concurrency model, and EPSM. Section 4 covers Tsavorite in detail, with durability covered in Section 5. Garnet's powerful cluster mode is described in Section 6. We briefly cover other implementation details in Section 7, report evaluation results in Section 8, cover related work in Section 9, and conclude in Section 10.

2 BACKGROUND ON RESP

RESP is a simple, efficient, and human-readable wire protocol [52] for remote cache-stores. Commands are processed and responded to in FIFO order per user session (or connection). Each message consists of a prefix, followed by the data payload. For example, a simple string message is prefixed with a `+`, an integer with a `:`, and an error with a `-`. Binary-safe *bulk strings* begin with a `$`, followed by the length of the string in bytes, a `CRLF` (`\r\n`), the string data itself, and another `CRLF`. For example, a 4-byte string "test" is represented as `$4\r\ntest\r\n`.

At its core, a RESP server simply stores keys and values, but the values can hold a variety of data types. Table 2 shows some example commands; the full RESP command set can be accessed online [18]. The simplest value type is a *raw string* of bytes, and commands on strings include reads, (blind) updates, atomic modifications, and time-based expiration. Values can also be complex data structures such as `List`, `SortedSet`, and `Hash`. Objects are treated as (remote) heap data structures: specific elements within the object (or object statistics) can be read, and an object's internal elements can be updated in a variety of ways. Further, operations such as `RPOPLPUSH` work transactionally across more than one object instance.

3 GARNET SYSTEM ARCHITECTURE

Figure 1 depicts the system architecture of standalone Garnet.

3.1 Network Layer

Garnet exposes network endpoints, which can be TCP sockets, Unix domain sockets, or user-defined endpoints such as eRPC [23] (with support for kernel bypass networking stacks such as DPDK and RDMA). Remote clients establish *sessions* with the Garnet endpoint, optionally performing a TLS handshake for encrypted connections.

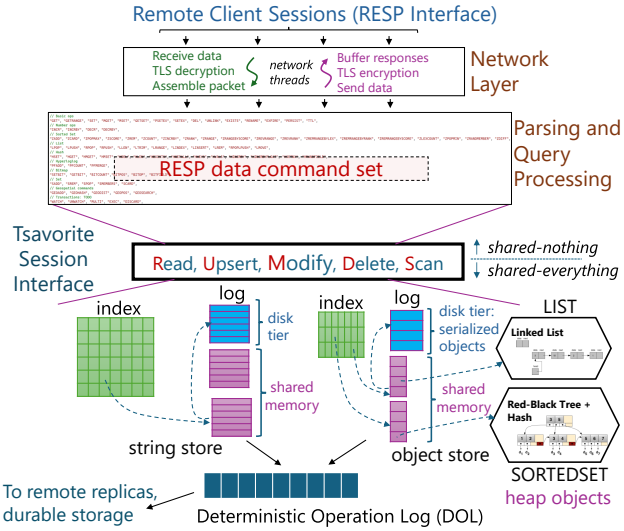


Figure 1: Garnet system architecture (single node).

Garnet sessions are strictly FIFO: the server receives a stream of request commands and produces a pipelined stream of responses. Network buffers in a session are processed by the thread that receives it, for instance, the OS thread pool thread that handles the network interrupt or the thread chosen by the kernel bypass library. This thread performs TLS decryption, and hands over the packet to the parsing and query processing logic of the Garnet session.

3.2 Parsing and Query Processing

An incoming (decrypted) network packet on the session contains a batch of RESP commands. We first parse each command using an optimized predictive parser that prioritizes the most common commands (such as `GET` and `SET`) and exploits the predictability of the RESP syntax to reduce the number of branches needed to validate the input. The parsed command, along with its arguments, is then passed on to the respective processor.

At startup, Garnet creates an instance of its storage engine (Tsavorite) and registers the supported command set. Tsavorite also supports the notion of FIFO sessions; every Garnet session holds a reference to its own Tsavorite session. Data commands are processed by invoking the narrow Tsavorite session interface called `RUMDS` (for `Read`, `Upsert`, `Modify`, `Delete`, `Scan`), passing in the command and its parameters as opaque input. All non-determinism (e.g., random numbers for commands such as `SPOP` that remove a random set member, or the current timestamp for expiration commands such as `SETEX`) is captured as part of the input parameters to the Tsavorite interface calls. We cover storage in Sections 4 and 5.

If durability is enabled, data commands that modify the store (e.g., string set or list insertion) cause an entry to be written to the DOL, a durable log of deterministic Tsavorite operations. The DOL is used for both server durability in standalone mode and keeping secondary replicas up-to-date in cluster mode (Sections 5 and 6).

3.3 Concurrency Model and EPSM

Concurrency in Garnet is corralled to stay within the internals of Tsavorite: multiple shared-nothing sessions of Garnet and Tsavorite operate on one shared-everything data store (index, memory,

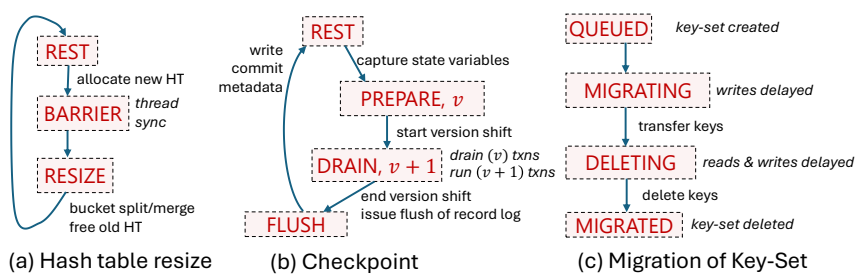


Figure 2: Epoch-protected state machine (EPSM) examples.

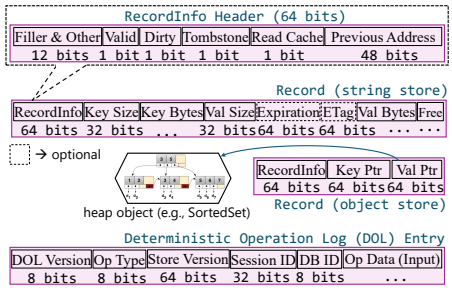


Figure 3: Tsavorite headers.

and tiered storage) per node. This design keeps the layers above Tsavorite (such as TLS processing and parsing) thread-safe and task-parallel by construction and provides session isolation, without loss of performance and scalability. Overall, a batch of commands in a single network packet on a session execute without any thread-switching (except for disk IO). This design ensures that registers and L1/L2 caches are optimally utilized for session state.

We introduce an *epoch-protected state machine* (*EPSM*) for infrequent yet concurrent control plane operations such as hash table growth, checkpoints, safe page flush and eviction, and key migration during scale-out. With EPSM, the system starts at the REST phase and goes through a sequence of phases. One can register custom control plane logic, called *actions*, to execute before a phase change. After a phase change, EPSM increments (or *bumps*) a global epoch counter [27, 33] from E to $E + 1$. Any thread performing store operations first enters the current epoch and reads the global EPSM phase. It can then safely operate assuming that the phase is stable until it completes the operation and leaves the epoch. When epoch E is safe to reclaim, i.e., all threads have progressed beyond epoch E and entered the new phase, EPSM can execute the next set of actions and move to the next phase. Examples of EPSM use are summarized in Figure 2 and discussed in subsequent sections.

Traditional epoch-based control planes [31, 43] let application threads cooperatively perform actions such as page flushes during epoch refresh. However, this can increase the latency for unlucky threads that need to complete user operations. This is solved in Garnet by using a background task that drives EPSM, so that application threads do not need to perform the actual EPSM actions.

4 THE TSAVORITE STORAGE LAYER

Tsavorite serves as the key-value storage layer of Garnet. Overall, it consists of two stores: a *string store* dedicated to string (or byte*) values (including bitmaps and HyperLogLog), and an *object store* dedicated to storing heap objects. This dual-store architecture allows us to optimize storage for each value type separately. Keys are always strings. Each store consists of a latch-free concurrent hash index over a *record log* of key-value pairs that span main memory and storage tiers. The record log is organized as a sequence of pages across storage and main memory, and the index directly points to the logical addresses of records in these pages.

4.1 Data Model

Records in Tsavorite—see Figure 3—start with a 64-bit record header that stores a 48-bit logical address of the previous record in the hash chain. The remaining bits are used to indicate various record properties such as the existence of optional fields in the record, tombstoned (deleted) records, and dirty records. This is followed by the key, which is simply stored as a 4-byte key length followed by the actual key bytes. For the string store, the value is stored in the same format as the key. For the object store, the value in memory is an 8-byte pointer used to de-reference the heap object, while the same value on disk is a sequence of bytes corresponding to the serialized object (similar to values in the string store). Large string keys and values in memory may also be stored on the heap, similar to objects. Values are optionally associated with a 64-bit expiration timestamp and/or a 64-bit *entity tag* (or ETag)—these are covered below. Finally, any remaining empty space at the end of the record is marked with an optional 32-bit filler length (Section 4.7).

4.2 Single-Key RUMDS Interface

Each operation in Tsavorite is designed as a pair of concepts: a front-end *interface* and a back-end set of *callbacks* for that interface. Garnet's query processor registers the callbacks that perform the actual storage interactions for each supported RESP command. At runtime, it simply invokes the corresponding front-end interface with input parameters, which eventually leads to Tsavorite invoking callbacks as necessary.

Tsavorite exposes five basic operations: Read, Upsert, Modify, Delete, and Scan. We call this the RUMDS interface, and find that it is sufficiently powerful to not only express hundreds of existing RESP data commands, but also new classes of commands (Section 4.3). Table 3 shows examples of mapping RESP to RUMDS. Note that in Garnet, Tsavorite can conceptually be replaced by any storage layer that implements RUMDS. Further, Tsavorite, using RUMDS, may be used directly (without the rest of Garnet) to handle functionality beyond RESP. Both these extensions are outside the scope of this paper. The RUMDS interface and callback semantics are described next; our latch-free concurrent implementation is covered in Section 4.4.

Read. The Read interface accepts a key, an *input*, and an *output* as parameters. Tsavorite searches the store for the value corresponding to that key. If such an unexpired value does not exist, it returns NotFound. If it does exist, Tsavorite invokes a registered callback called Reader—application (i.e., Garnet) logic whose goal is to *transform* the value into the output.

In case of Garnet, input is the RESP command ID and output is usually the network response buffer. Depending on the RESP command, Garnet's Reader performs different actions, for example:

(1) GET writes the value as a RESP bulk string to the output.

(2) GETRANGE copies the specified value sub-string to the output.
(3) TTL writes the value's expiration in the RESP number format.

Note that the callback logic is *declarative*—its implementer does not know whether the value was found in main memory or disk.

Upsert. The Upsert interface accepts a key and an input as parameters, and is responsible for inserting a record into the store. It is a *blind* operation, i.e., it does not need to incur an I/O to retrieve any older record from disk. The application provides several callbacks for Upsert:

- InPlaceWriter: This callback is responsible for overwriting an existing record in-place if possible, returning false if not possible, e.g., the new value did not fit in the available space.
- InitialWriter: This callback is responsible for writing a newly allocated record of the requested size—determined by a callback called GetLength—using the specified input.

In case of Garnet, input is the RESP command ID, the value from the network request buffer, and optional fields such as expiration times. Depending on the RESP command, Garnet's SingleWriter and ConcurrentWriter perform actions such as copying the value from the network buffer directly into the store's record in case of SET, and adding an expiration field to the record in case of SETEX.

Modify. The Modify interface also accepts a key and an input as parameters. This powerful operation logically represents the evolution of a record over time, from its initial creation to its eventual deletion. Operations on objects such as List and SortedSet are implemented exclusively using this interface. Most generally, it stands for atomic read-modify-write, although because it is based on callbacks, specific phases of the operation may be elided. The following callbacks are associated with Modify:

- NeedInitialUpdate: This callback returns whether the command should evolve the record from a non-existent state. Garnet uses this callback to return false in case of RESP commands such as SETXX, where we want to atomically update a record only if it already exists in the store.
- InitialUpdater: This is the initial value based on input. For INCRBY, it sets the initial value to the specified delta. Objects are initially created only by this callback, e.g., during an LPUSH (list push) command on a key that does not yet exist in the store.
- InPlaceUpdater: A record in memory is evolved in-place using this callback, which accepts a pointer to the existing object or string in order to update it in-place based on its existing value and the specified input. For example, INCRBY on a value of say 85 with an input delta of 10 would atomically evolve the value to become 95. A second INCRBY of delta 10 may not fit in-place since 105 occupies an extra digit. In this case, the callback returns false and control flows to NeedCopyUpdate, described next.
- NeedCopyUpdate: This callback determines, after reading the old value from disk, whether we need to proceed with the operation. This usually returns true, but is false in case of commands such as SETNX (set if not exists), as the existence of a previous value for the key implies that we should not evolve it further.
- CopyUpdater: This callback evolves the record's value based on the specified prior state and specified input. For example, INCRBY on a value of say 95 with an input delta of 10 would allocate the necessary space—determined by a GetLength callback—and atomically evolve the value to become 105.

Table 3: Examples of mapping RESP commands to RUMDS.

RESP Cmd	RUMDS Op	Callback	Action
GET key	Read	Reader	Copy value for the key from storage to buffer in RESP bulk string response format.
SET key value	Upsert	InPlaceWriter	Overwrite value over existing one, return true if there was space, false if not.
		GetLength	Return length of input value.
		InitialWriter	Copy value to the allocated space.
SETXX key value *(set if exists)*	Modify (string)	NeedInitialUpdate	Return false.
		InitialUpdater	– n/a –
		InPlaceUpdater	Overwrite value over existing one, return true if there was space, false if not.
		NeedCopyUpdate	Return true.
		GetLength	Return length of input value.
		CopyUpdater	Copy value to the allocated space.
SETNX key value *(set if not exists)*	Modify (string)	NeedInitialUpdate	Return true.
		InitialUpdater	Copy value to the allocated space.
		InPlaceUpdater	Do not copy, return true.
		NeedCopyUpdate	Return false.
		GetLength	– n/a –
		CopyUpdater	– n/a –
LPUSH key value	Modify (object)	NeedInitialUpdate	Return true.
		InitialUpdater	Create new List with value, on the heap.
		InPlaceUpdater	Push value to existing list, return true.
		NeedCopyUpdate	Return true.
		GetLength	8 bytes for heap objects (pointers).
		CopyUpdater	Create new list with old contents, push value.

Delete. The Delete interface is similar to Upsert—it adds a tombstone record, or sets a tombstone bit in an existing record if it could be updated in place. We omit the details for brevity.

Scan. This interface performs a scan of the store's snapshot and issues callbacks for each live record. It is used for operations such as DBSIZE and SCAN. DBSIZE returns the total number of keys in the system, and is implemented by a Scan with a callback that simply increments a counter. KEYS [pattern] returns keys that match a specified optional pattern; we again use a Scan with a callback that performs pattern matching using the standard regular expression library and writes matching keys to the network response buffer.

4.3 Using RUMDS Beyond RESP

The RUMDS interface has allowed Garnet to expand beyond the RESP command set. For example, records can store an optional 64-bit ETag, which can be updated and checked as part of RUMDS operations. ETags have allowed contributors to add commands that facilitate new use cases. The SETIFGREATER command updates a value only if the provided ETag is greater that the current one. This can ensure consistency when multiple clients make updates to a database and a front-end cache; cache updates succeed only if the resource has not been modified elsewhere. The GETIFNOTMATCH command retrieves a locally cached value only it was modified at the remote cache, avoiding network traffic when values are unchanged. Such extensions are out-of-scope for this paper; see [17] for details.

4.4 Concurrent Data Structure

Tsavorite's core building block is a *store*: a data structure that consists of a record log spanning main memory and storage, accessed using a hash index. We maintain two stores: the *string store* holds strings whereas the *object store* holds objects. These components are described next (see Figure 1).

Record Log. The record log consists of a disk portion and a memory portion. The disk portion is addressed using append-only 48-bit logical addresses starting from 0. Main memory is organized as a circular buffer of pages, and are addressed by simply concatenated (page-id, offset) pairs. The in-memory pages are fixed size and contain records. In case of the string store, keys and values are

bytes usually stored in-line in the page. The object store is based on string keys, but the values are pointers to out-of-line heap objects (e.g., red-black trees for `SortedSet`), which enables fast object modification and read operations in memory. Older pages in the circular buffer are marked read-only and are eventually flushed to disk. Before evicting the oldest (flushed) page in the circular buffer, heap objects are serialized and hash chains for records on the page are updated with disk addresses. We use EPSM to ensure that pages are safe to flush or evict in the presence of concurrent reader and writer threads.

The disk portion of the record log is based on `IDevice`, a device abstraction for log-structured store [53] operations, i.e., sequential tail writes and random reads. We implement efficient devices for local disk and cloud storage. Tiering is achieved using an abstraction we call a *tiered storage device*—this is simply a logical `IDevice` implementation that holds N real device implementations. Reads are served by the highest level that has the data, and writes are written through to all the tiers.

Hash Index. Tsavorite indexes the tiered record log using a latch-free hash index consisting of a power-of-2 number of 64-byte cache-line sized buckets. Each bucket contains seven 8-byte bucket entries. Each entry in a bucket is associated with a *tag* (t bits from the key hash) and a 48-bit *logical address* into the record log. The address serves as the root of a reverse-linked-list hash chain that threads every colliding record, i.e., the record's key hash bits map to that bucket and tag. For example, with a 2^8 buckets and $t = 4$ tag bits, we would use the first 8 bits of the key's hash to choose the bucket. The entry for this key would store a tag corresponding to the next 4 bits of the hash. We adapt prior two-phase latch-free algorithms [5] to ensure that every bucket has at most one entry per unique tag. The last (eighth) entry in a hash bucket is special: it stores a 16-bit *bucket lock* along with a 48-bit address to an overflow bucket (if present). This design allows the lock table to be accessed without incurring additional cache misses after a hash lookup.

An EPSM—see Figure 2(a)—is used to resize the hash table. We allocate the new hash table and enter the `BARRIER` phase, which causes threads to synchronize for the upcoming size change. The next `RESIZE` phase consists of threads operating on the new hash table, re-hashing old buckets as necessary. Once all buckets are transferred, the old table is freed and we end at the `REST` phase.

4.5 Concurrent `RUMDS` Implementation

The `RUMDS` interface is heavily used for most cache-store operations, and needs to run at high performance in a thread-scalable manner. `Read` follows the hash chain, first through memory and then on disk. A full key comparison is made to ensure that we have found the requested key. If found, the `Reader` callback is invoked under a shared bucket lock. `Upsert` and `Modify` operations start similarly, with the difference that `Upsert` limits its search to the mutable region in memory. If the record is found in the mutable region, modifications are attempted in-place on the record log values using `InPlaceWriter` (for `Upsert`) and `InPlaceUpdater` (for `Modify`) callbacks under an exclusive bucket lock. Otherwise, we need to add a new entry to the hash chain.

Tsavorite uses latch-free structure modifications, i.e., all changes to the hash entry and chain are based on the notion of optimistic local modification, followed by an atomic `compare-and-swap` (CAS) to install the update for global visibility. For example, when `Upsert` or `Modify` needs to add a record to the hash chain, it allocates space at the tail of the record log using `fetch-and-add`, creates the record using the `InitialWriter` (for `Upsert`) or `InitialUpdater` and `CopyUpdater` (for `Modify`) callback, and finally updates the hash chain using a CAS at the hash entry. On CAS failure, the space is marked for reuse and the operation is retried. Since the callbacks above were optimistically issued before the CAS, additional callbacks (`PostInitialWriter`, `PostInitialUpdater`, and `PostCopyUpdater`) are provided for work that needs to happen *after* a successful modification, e.g., writing to the DOL. Note that only structure modifications are latch-free in Tsavorite; application-level shared and exclusive bucket locks (Section 4.4) are still necessary for concurrent reads and updates of records.

4.6 Multi-Key Transactions

Garnet supports multi-key transactions to handle (1) standard Redis transactions [49], which consist of a set of commands bracketed by `MULTI` and `EXEC`; (2) server-side Lua scripts [36, 48]; (3) transactional stored procedures; and (4) atomic cross-key commands such as `RPOPLPUSH` (Table 2). We achieve thread-scalable transaction processing by building on Tsavorite's locking primitives.

Garnet's concurrency control for transactions is a hybrid model of *pessimistic two-phase locking (2PL) with optimistic watching*, designed to be efficient for workloads dominated by single-key operations and rare conflicts. It works orthogonally to EPSM, which is only used to ensure transactionally consistent snapshots during a checkpoint (see Section 5.1). As running example, consider a transaction that transfers 10 units from key `src` to key `dest`:

```
MULTI
  DECRBY src 10
  INCRBY dest 10
EXEC
```

Pessimistic Locking. Before the transaction starts, we identify the read/write set of keys by parsing the queued commands between `MULTI` and `EXEC`. Since `INCRBY` and `DECRBY` are `Modify` operations, the read and write sets are both {`src`, `dest`} in our example above. We then acquire hash bucket-level shared/exclusive locks for the read/write set in sorted bucket order for these keys, to prevent deadlocks. Finally, the commands are executed and the locks are released. Transactions are optionally logged for durability in the DOL (see Section 5.2).

Optimistic Watching. Keys may be *watched* [59] for modifications before the transaction. Watches are a form of optimistic concurrency control, where one can watch, then read, certain keys even before the transaction starts. These values can be optimistically used as parameters in the actual transaction. For example we may watch, then read, a key `amount` in our running example to use as the transfer amount instead of the hard-coded value of 10 units:

```
WATCH amount
amt = GET amount
MULTI
  DECRBY src $amt
  INCRBY dest $amt
EXEC
```

All watched keys are added to the read set. After all locks are acquired, if we find that any watched key (e.g., `amount`) was modified post-`WATCH`, the transaction is aborted. To detect such modifications

of watched keys, we use a *watch table*, a small shared array of 64-bit version numbers. A write operation on a watched record increments the version number of the watch table entry corresponding to its key hash (modulo the size of the table). When the `WATCH` command is issued for a key, we store the current version number of that key's slot as part of the transaction state. After lock acquisition, we check whether any watched slots have a larger version number. If yes, we abort the transaction due to a (potential) conflict. Otherwise, transaction processing proceeds as described earlier.

4.7 Memory Optimizations

Tsavorite is designed to be a larger-than-memory cache-store, but optimized when the working set fits in main memory. Log-structured designs generally suffer from space amplification, which is alleviated in Tsavorite by in-place updates in the mutable region of memory. However, there is wasted space in memory due to record deletion, failed in-place-updates (e.g., because the new value was larger), failed CAS operations, and expired records.

Expiration. Records logically expire when time moves past the stored expiration time. However, they stay in the log and occupy space. For expired main memory records, we use a background thread to find and make the space available for future insertions using a technique called revivification (Section 4.7). For expired and deleted disk records, log compaction (periodically identifying live records on the oldest pages and writing them to the tail) is used to eliminate dead records from disk and reclaim disk space.

Revivification. When records slots in main memory become invalid, e.g., due to record expiration or deletion, they waste space. Revivification aims to make such space available for reuse by future write operations. This allows memory to be used judiciously and may also relieve contention at the tail and unnecessary log growth. Revivification primarily involves maintaining a *free list*—a binned set of circular buffers that hold pointers to reusable record slots. A deleted record is eligible for safe elision from the hash chain if we can determine that no earlier record with the same key exists in the chain. Such records are elided from the hash chain and added to the free list. We use epoch protection to make sure that the record is safe from any further concurrent accesses. When a new record slot is required, the system attempts to use an available safe record slot of the closest size from the free list.

Read Cache. Records that are read frequently need to be retained in memory. Garnet uses a *read cache*—another in-memory record log that resides between the hash index and the main record log—to hold read hot records. The hash chain links from the index to the optional read cache, and then to the main record log.

5 DURABILITY

Large applications and services typically start by deploying caches in front of a durable database. Such deployments do not require durability, but "best effort" durability may be necessary. Cached state may need to be saved (by taking a checkpoint) in order to allow restart with a warm cache. Applications may desire to eliminate the overhead of having two (sometimes diverging) data sources, and want the caching layer to offer durability guarantees [57]. Durability should have minimal impact on tail latencies and memory footprint during normal operations. Below, we overview the efficient and tunable durability mechanisms in Garnet.

5.1 Checkpointing

RESP clients can request a `SAVE` operation, which translates to a checkpoint of the store. Redis uses a process `fork` mechanism for taking the checkpoint. This is expensive since updates during the checkpoint can (in the worst case) double the memory footprint of the node and cause severe slowdown in practice [57]. In a multi-threaded system like Garnet, all threads would need to momentarily halt for the `fork` in order to get a consistent checkpoint.

Garnet uses a non-blocking checkpoint mechanism called *snapshot*, inspired by our prior work on concurrent prefix recovery (CPR) [43]. Briefly, CPR allows a database to take a consistent checkpoint without blocking individual threads by coordinating to move the database from the old version v to the new version $v + 1$. Garnet uses EPSM—see Figure 2(b)—to implement snapshots: a prefix-consistent snapshot is obtained by restricting $v + 1$ threads (`DRAIN` and `FLUSH` phases) from updating unflushed v records in place, while v records are flushed to disk as the `FLUSH` action. However, two new complications arise in Garnet: (1) Garnet needs to create a consistent checkpoint across two stores (the string store and the object store), not one; (2) Garnet needs to handle transactions over more than one key, and across the two stores.

To solve the first problem, we decouple the state machine from the store, allowing more than one store to be registered to the same state machine. The state machine stores the Garnet database version number, and drives the CPR checkpoint phases uniformly against both stores simultaneously. The atomic transition from version v to $v+1$ is common across the stores, resulting in a consistent snapshot. We believe this to be a general extension that can be used to take consistent snapshots in a multi-database setting as well.

One way to solve the second problem is to drain out active (v) transactions at the version transition point. This would however cause a hiccup in latency due to blocking. We make the observation that because Garnet uses pessimistic locking, we have the opportunity to decide a transaction's version (v or $v + 1$) *after* its locks are acquired. This is safe because there is no longer the risk of a $v + 1$ transaction updating a value that a v transaction is also updating, or of a v transaction reading a value that a $v + 1$ transaction has updated. Based on this insight, we can move the state machine to $v + 1$ with a `DRAIN` phase that lets threads run in either version concurrently. Once all the v transactions have completed, the v version is safe to snapshot and we move to `FLUSH`, where pages are flushed for a consistent checkpoint. We note that conflicting transactions will indeed block as usual, but this is fundamental and unrelated to the checkpoint mechanism.

5.2 Deterministic Operation Logging

Single-key write operations are logged in a unified deterministic operation log (DOL) in Garnet. Each DOL entry has the format shown in Figure 3. Instead of logging the user's RESP commands, we log at the `RUMDS` interface of Tsavorite, where we have a tightly packed operation input that includes the encoded RESP command, parameters, a database version to correctly skip operations that were part of a recovered recovered checkpoint, and fields to make operations deterministic, such as timestamps and random numbers. Transactions add special entries to the DOL after lock acquisition and before transaction completion; they enable our recovery and replay logic to re-apply the transactions in the correct serial order.

Garnet's DOL is implemented using a shared circular memory buffer of pages with `fetch-and-add` on the tail address. Commit is issued periodically in the background or after every operation, depending on the level of durability required. The DOL performs the group commit optimization, and write operations have the option to either return immediately or wait for the DOL to commit first.

The device we flush to can be a tiered device, e.g., with SSD and cloud storage. Writes are propagated to all tiers, but we may identify some tier to indicate commit. For example, we may commit to SSD (say with 100μs latency) and cloud storage (say with 5ms latency), but acknowledge the write as soon as the SSD completes the flush. This configuration lets the cache-store fully survive node restarts and partially survive a fresh start on a new node.

6 GARNET CLUSTER DESIGN

Garnet cluster provides the ability to operate Garnet across multiple nodes in a distributed system. It supports scale-out through data sharding, and dynamic scaling though re-sharding with data migration for availability, fault tolerance via replication.

6.1 Cluster Architecture Overview

Garnet clusters consist of nodes that manage client access to data through a common configuration. Following the RESP protocol, every shared-nothing node operates in a virtual key space that is partitioned into 16,384 hash slots. Data access is determined by mapping keys to specific slots using a CRC16 [42] hash function. The nodes rely on the slot assignment information to serve requests or redirect them to the original owner. Multi-key operations are restricted to operate on keys within the same slot.

The nodes in a cluster can be characterized by assuming the role of a *primary* or a *secondary*. Primary nodes are owners of a subset of all slots and can serve both read and write requests. They are also responsible for coordinating data re-sharding during scale-out operations. Secondary nodes are associated with a single primary and can serve only read requests. A primary node can have any number of secondary nodes, usually as many as required to achieve the desired level of redundancy or read throughput.

Garnet offers a RESP control interface for cluster that enables updates to the cluster configuration at runtime. This interface responds to events such as adding primary nodes (to scale out) and secondary nodes (for redundancy and read throughput), and promoting a secondary to a primary during failover. Garnet handles the propagation of configuration changes via a gossip protocol.

6.2 HashSlot Migration

When a hash slot is reassigned to a new node, we need to perform *HashSlot migration*, which refers to moving all keys in the slot to its new owner. Redis relies on the client for HashSlot migration, but this results in significant network traffic. Instead, Garnet implements HashSlot migration using direct server-to-server communication which improves performance and ensures availability.

The Redis cluster specification [45] outlines an API to orchestrate slot migration. This API relies on tracking the state of each slot within the node's local configuration snapshot. At first, all slots are assigned to distinct primaries, and their state is set to `STABLE`. During migration, the state of a given slot is set to `IMPORTING` at the target node and `MIGRATING` at the source node. This change in state modifies the way the server responds to client requests. Read and write requests are served if the key exists; otherwise, the node redirects the client to the target node with a special `ASK` response. At the source node, a RESP command (`MIGRATE`) is used to trigger the transfer of key-value pairs that belong to the slot. The target node assumes ownership of the slot by transitioning its state to `STABLE` after data transfer is complete. We use EPSM (Section 3.3) to implement these slot-level state changes in Garnet.

Algorithm 1 Migration algorithm.

```
S ← Slots to be migrated
M ← Migration key-set
S ← Source node        T ← Target node
for X ∈ S do
    EPSM.TransitionSlotRemote(X, T, IMPORTING)
    EPSM.TransitionSlotLocal(X, T, MIGRATING)
    while M = ScanForNextBatch(X, QUEUED) do
        EPSM.TransitionKeySet(M, MIGRATING)
        Send(M)
        EPSM.TransitionKeySet(M, DELETING)
        DeleteKeys(M)
        EPSM.TransitionKeySet(M, MIGRATED)
    end while
    EPSM.TransitionSlotLocal(X, T, STABLE)
    EPSM.TransitionSlotRemote(X, T, STABLE)
end for
```

We require that Garnet `RUMDS` operations perform their key existence check and subsequent processing concurrently with migration. Using slot-level locks would severely impact throughput and availability. Our solution involves another EPSM—see Figure 2(c)—this time, for each *key-set*. A key-set tracks a small batch of keys that are migrated together, using a compact Bloom filter [3]. The key-set starts in the `QUEUED` phase. Phases indicate different levels of concurrent access to keys in the key-set by operations. EPSM ensures that concurrent operations on the store observe the relevant transition before we take action, eliminating the need for locking.

In the fast path, keys that do not exist in the key-set can be operated on as usual. The same holds true for keys in the `QUEUED`[3] state. Once a key-set transitions to the `MIGRATING` state new write requests are delayed (for keys in the key-set) but read operations and read-only transactions can continue. State transitions are managed using EPSM which ensures that ongoing transactions and operations that overlap with the key-set are completed before transmitting the relevant data to the target node. Once data transmission completes, the key-set enters the `DELETING` state, where keys are removed from the source node's database. In this state, both reads and writes must be delayed in order to maintain consistency. Finally, upon deletion of the key batch, the key-set enters the `MIGRATED`, signaling waiting operations to proceed Note that false-positives in the Bloom filter do not affect correctness because they only result in rare operation delays. Migration is summarized in Algorithm 1. The slot-level EPSM referenced earlier prohibits sessions from adding new keys during slot migration.

6.3 Replication

Garnet cluster supports an asynchronous replication protocol that is based on the primary-secondary model. At its core, the replication protocol relies on the DOL to stream batch updates to the attached

[3]The `QUEUED` state helps preserve availability, as scanning the database for relevant keys can be time-consuming.

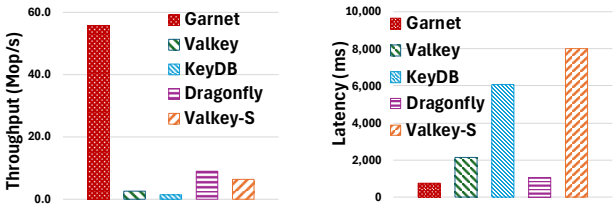
secondaries. Adding secondary nodes can happen at any point during runtime and is often an operation initiated by the cluster operator. Once such an operation is initiated, the secondary begins with transmitting the relevant metadata to the primary. Using the transmitted information, the primary has to decide to perform either a full or partial synchronization. This decision is guided by the primary's ability to serve the secondary's request using an intact DOL address space. In many cases, this may not be feasible because the DOL has been truncated following an earlier snapshot to conserve disk space, or the in-memory DOL buffers have been overwritten due to capacity constraints. If a full synchronization is necessary, the primary has to take a full snapshot and transmit that to the secondary, so it can recover before being ready to accept future DOL records. Note that the primary may proactively take an *on-demand checkpoint* to reduce network writes when the resulting snapshot is smaller than the cumulative size of the DOL records that would otherwise need to be transmitted.

Once recovery completes, the primary creates a dedicated background task to be used for streaming DOL records to the secondary. The secondary then adds these records to its local DOL, whose contents are kept identical to the primary DOL in order to facilitate future promotions to primary. An asynchronous task scans the secondary node's DOL and applies the `Upsert` and `Modify` operations on Tsavorite. Note that during a concurrent checkpoint, the DOL may have v and $v+1$ entries interspersed in the log. Since each DOL record has a database version, secondary replay can skip records that were part of the recovered checkpoint, ensuring that every operation is applied exactly once at the secondary.

Streaming Snapshot. Full synchronization usually requires taking a checkpoint at the primary and shipping it to the new secondary. This is an expensive operation involving the disk at the primary. We introduce a new checkpoint variant called a *streaming snapshot* that overcomes this limitation. The key idea is to stream database records that together denote a consistent point-in-time snapshot of the store, without blocking the system from ongoing operations or interacting with disk. The overall logic uses an EPSM to orchestrate this procedure. In the first phase, we use `Scan` (from `RUMDS`) for the immutable region of the record log, and incrementally return a stream of live records (liveness is checked using the hash index). In the second phase, we use EPSM analogous to the checkpoint EPSM that transitions the database from v to $v+1$. However, unlike a traditional checkpoint, our `FLUSH` phase does a `Scan` through the mutable region of the record log and returns all live v records. The protocol guarantees that concurrent $v+1$ updates are not applied in-place, thereby ensuring point-in-time snapshot semantics. As an optimization, an update operation during the `FLUSH` phase can directly add live v records to the iterator instead of performing a read-copy-update; this avoids the record log's tail growth during the streaming snapshot. Note that records returned in the second phase may rarely overlap with records from the first phase; in these cases, the latest record in the stream represents the consistent value for the snapshot.

7 IMPLEMENTATION DETAILS

Garnet was implemented in C# after a careful analysis of the available options. We wanted a high-level language which offers a rich library of data structures that one could use to quickly implement the rich command set of RESP. At the same time, the language had to allow us to write extremely low-level code (e.g., avoiding garbage collection, using pointer based operations and SIMD instructions). C# was a good match with its flexibility of object-oriented programming, developer familiarity to encourage open-source contributions, ability to work across numerous hardware architectures (Intel, AMD, ARM) and operating systems (Linux, Windows, Android, MacOS), and friendliness to writing low-level system code. RESP clients are available in almost every programming language, so the choice of server language did not affect the system's broad usability. Garnet supports most RESP commands today; the documentation [18] has more details.



(a) Memtier throughput. **(b) Memtier latency (P99).**

Figure 4: Memtier benchmark results.

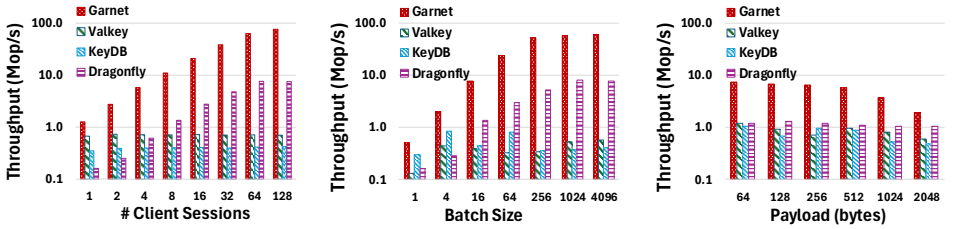
Garnet supports stored procedures and modules. These can be registered dynamically, and cover four classes: (1) Single string operations; (2) New object types and operations over them; (3) Stored procedures—with and without transactions; and (4) Lua scripts. Existing RESP clients can be used to invoke these commands. Further, Garnet supports modules, which encapsulate server-side logic that packages and registers these extensibility options.

8 EVALUATION

We evaluate Garnet in two ways. First, we compare end-to-end throughput, latency, and durability in standalone mode, varying various workload and system attributes. Next, we evaluate Garnet in cluster mode, focusing on migration and replication.

8.1 Setup and Workload

We provision several Azure Standard F72s v2 virtual machines (72 virtual cpus, 144GB memory each) running Linux (Ubuntu 24.04), with accelerated TCP networking [16] enabled. One machine serves as the client and others run as cache-store servers in standalone and cluster mode. We use two load generator clients (`memtier_benchmark` [51] and `Resp.bench` [58]) to compare the performance of Garnet to the latest open-source versions of Valkey (v8.1), KeyDB (v6.3.4), and Dragonfly (v1.29.0). We also evaluate Valkey with intra-node sharding (Valkey-S) using the cluster mode to utilize all cores. We found open-source Redis to perform similarly to Valkey, since the latter is a fork of the former, and do not include it in most results. Workloads are based on a RESP translation of the YCSB-A workload from the Yahoo Cloud Serving Benchmark [7], with 256 million distinct 8-byte keys, string values ranging in size from 8 bytes to several KB, and complex objects.



(a) Varying clients (256M keys). (b) Varying batch size (1M keys). (c) Varying payload size (1M keys).

Figure 5: GET throughput experiments (Uniform Distribution).

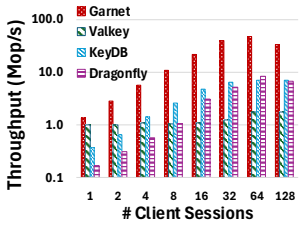


Figure 6: SET throughput experiments (Zipf distribution).

8.2 Memtier Benchmark Results

We run `memtier_benchmark` [51], an open-source RESP load generation tool used widely to compare cache-stores. We use a 1:9 SET-to-GET ratio, using a keyspace of 256 million keys and an 8-byte payload. To stress the systems, we use a large number (6400) of clients, using 128 threads with 50 clients per thread. Figures 4a and 4b show the measured throughput and P99 latency, respectively. We see that Garnet achieves at least 20× higher throughput and 3× lower latency compared to standalone Valkey. While intra-node sharding with Valkey-S narrows this gap, it increases latency due to sharding overheads. Garnet also outperforms multi-threaded KeyDB and Dragonfly, delivering up to 40× and 6× higher throughput respectively. A similar trend is observed for P99 latency at high load, with Dragonfly the closest: about 1.3× slower than Garnet.

We found that `memtier_benchmark` causes the client to become the bottleneck at very high throughput, so the rest of the experiments use `Resp.bench` [58]–our custom-built lightweight tool that uses a cache of pre-generated client commands—to ensure that the client does not limit the reported server performance.

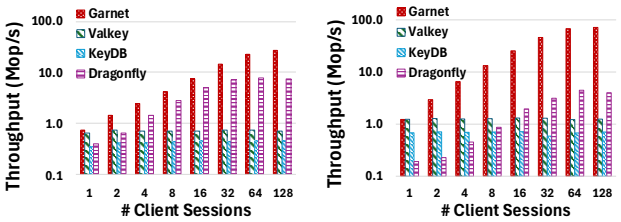
8.3 Standalone Throughput

String Operations. We measured the throughput of GET operations by varying payload size, batch size, and number of client sessions over a pre-loaded database; see Figure 5.

With an increasing number of client sessions (Figure 5a), we observe that Garnet exhibits better scalability than Valkey with 108× better throughput (note that the y-axis is log scale). KeyDB performs worse than Valkey due to the global lock contention. Dragonfly scales well up to 16 threads, but has 10× lower throughput than Garnet with 128 clients. Note that Dragonfly is a pure in-memory system. Even for small batch sizes (see Figure 5b), Garnet outperforms the competing systems. Finally, Garnet is efficient for larger payload sizes with a small batch size (16) as shown in Figure 5c.

As a remote cache-store, Garnet saturates the client-server network bandwidth before hitting server-side memory bandwidth limits. We measured the available network bandwidth using the `iperf` [14] tool to be 30Gbit/sec. Garnet saturated this bandwidth with a 64 byte payload size, when using a batch size of 1024. This includes the RESP protocol overhead of around 11% (7 bytes for the 64 byte value). Larger payload sizes proportionally reduced the batch size needed to saturate network bandwidth, as expected.

In Figure 6, we measure throughput with a 100% SET workload with a Zipf distribution ($\alpha = 0.9$). Garnet performs and scales much better than other systems. As expected, the contention due to skew affects all systems at high thread counts. Dragonfly outperforms Valkey and KeyDB, but is significantly worse than Garnet.



(a) Varying clients (ZADD, ZREM). (b) Varying clients (PFADD).

Figure 7: Complex object throughput.

Complex Data Type Operations. Next, we benchmark performance of operations over complex data types, focusing on `SortedSet` (`ZADD`/`ZREM`) and HyperLogLog (`PFADD`, which adds a key to the sketch). Figure 7a shows the results for `SortedSet`, with 1024 objects and each operation updating a random object. In spite of more time being spent in object update, Garnet outperforms the other systems and scales well. Valkey and KeyDB achieve 1.5× to 40× lower throughput. Although, Dragonfly fares better, it does not scale as well as Garnet, achieving 1.8× to 3.5× lower throughput.

In Figure 7b, we evaluate the HyperLogLog `PFADD` command, adding keys to a randomly chosen sketch out of a million. As before, Garnet outperforms the competition when a large number of sessions is used. Our implementation uses hardware intrinsics to accelerate hashing when adding to the HyperLogLog sketch. This adds to the scaling capabilities of Garnet which results in outperforming its closest competitor by a factor of 10×.

8.4 Standalone Latency

Our latency experiments were performed on an empty database and for a combined workload issuing a mixture of 80% GET and 20% commands that operate on a small keyspace (1024 keys). Since we care about latency, our DB size is kept small while we vary other parameters of the workload such as client threads, batch size, and payload size. For every varying experimental parameter, we fix the value of the other two as indicated below:

Sessions	Batch Size	Payload Size (bytes)
1 – 128	1	8
128	1 – 64	8
128	16	64 – 2048

Our experiments concentrate on the P99.9 latency (Figure 8) averaged across multiple runs. Garnet consistently delivers lower and more stable latency compared to other systems as the number of

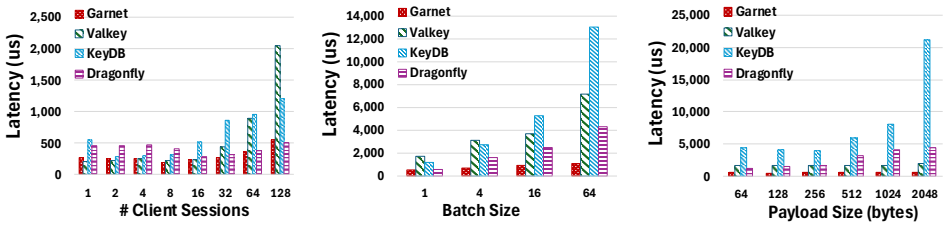


Figure 8: P99.9 GET(80%)-SET(20%) latency for varying parameters.

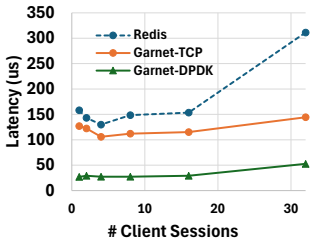


Figure 9: P99 PING latency.

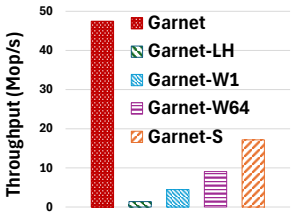


Figure 10: Design study.

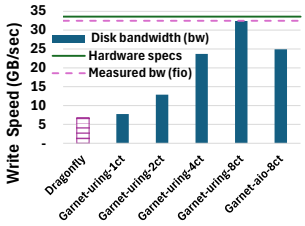


Figure 11: Durability.

clients increases. Designed to leverage adaptive client-side batching, Garnet outperforms competing systems by an even wider margin with batched input. While Dragonfly handles medium batch sizes more effectively, it struggles to sustain that performance at larger batch sizes. With increasing payload size, we observe that Garnet continues to perform well—for large payloads, the bottleneck for all systems becomes the available network bandwidth.

Kernel Bypass Networking. Garnet can work with different network implementations. We evaluated end-to-end latency at the 99th percentile, with hardware-accelerated TCP (for Redis and Garnet) and DPDK using eRPC [23] (for Garnet). The latter is a prototype, not part of open-source Garnet. Clients issue `PING` commands with no batching. We see from Figure 9 that Garnet exhibits higher stability at large percentile values. Further, as expected, kernel bypass provides a further 4 – 5× reduction in end-to-end latency.

8.5 Design Study

We study Garnet's chosen design by comparing it to alternatives (see Section 1.2 for details) that we implemented by modifying Garnet: (1) Garnet-LH, a design where networking is parallelized but storage is guarded by a lock; (2) Garnet-W1 (and -W64), in-process shards of 1 (and 64) storage *worker threads* that are used by our multi-threaded network layer; and (3) Garnet-S, which runs 64 independent Garnet processes on a single node, with clients handling request routing. We use `memtier_benchmark` as our client with 256M keys, 8-byte payloads, 64 threads with 1 client per thread, a batch size of 1024, and a uniform 1:9 SET-to-GET ratio.

Figure 10 shows the results. Garnet achieves a high throughput of 47Mops/sec. As expected, Garnet-LH does not scale and achieves poor throughput (1.3Mops/sec). Garnet-W1, which instead uses a single worker queue does better (4.4Mops/sec) due to better cache efficiency than a single contended lock. Garnet-W64 does better in aggregate (9Mops/sec), but does not scale due to the high overhead incurred by request splitting, data shuffling, and result collation. Garnet-S achieves better throughput (17.1Mops/sec), but this is because the request splitting and routing overheads are pushed to the client side, which is undesirable in practice. We also experimented with a skewed Zipf distribution (not shown) and found the skew to negatively impact Garnet-CQ64 and Garnet-S due to workload imbalance across threads, which the shared-everything design of Tsavorite avoids. To summarize, Garnet's chosen architecture (shared-nothing sessions operating over a shared-everything data store) significantly outperforms all the other considered designs.

8.6 Durability

We load 256 million keys (12-byte keys, 100-byte values) into each cache-store and perform a `SAVE` operation (see Figure 11). We configured storage using RAID [41] with 8 NVMe SSDs (4.2GB/s sustained write speed [10] each) for a total theoretical write speed of 33.6GB/s. The measured write speed using the `fio` [15] tool was 32.5GB/s. Garnet, using the `io_uring` [1] library and 8 IO completion threads (ct), took a checkpoint in 1.06s, saturating this measured limit. In contrast, Dragonfly was over 4× slower, taking 4.64s for a checkpoint. We also see that using fewer `io_uring` completion threads or the `libaio` [28] library, Garnet was not able to saturate the high available write bandwidth. Finally, Valkey took 273.1s to checkpoint the same data, and KeyDB was even slower, taking 481.7s.

8.7 Cluster Mode

Migration (Scale-Out). We demonstrate Garnet's ability to rapidly scale out with an experiment using two cluster nodes. We measure the total time required to migrate all slots from one node to the other. The results of this experiments are shown in Figure 12. The experiment assumed a preloaded cache with 256 million 8 byte keys and values. Garnet outperforms Valkey by 94× in terms of time taken to migrate all slots. This is largely because Valkey requires clients to migrate the data between the two nodes, which is extremely expensive. Similar to Garnet, Dragonfly does not require the client to migrate the actual data. However, Garnet is much faster, taking 4× to 8× less time to migrate all the data across nodes.

Replication. We evaluated the performance of replication by focusing on how well each system handles the overhead of both serving client requests and transferring the DOL to each replica.

Our first experiment (Figure 13) measures the achieved write throughput with increasing number of secondaries. To eliminate any other system overheads, apart from that of writing to the DOL and shipping it to the secondaries, we used 4 clients that perform 100% write on a keyspace of 1024 keys with 8 byte randomly generated payloads. Every write appends a new entry to the log (DOL) of each system, stressing their replication implementation. We see that Garnet outperforms other system and scales well even with 4 secondaries. Valkey and KeyDB sustain overall lower but stable

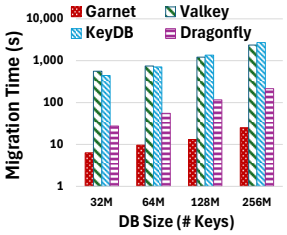


Figure 12: Migration time, varying batch size.

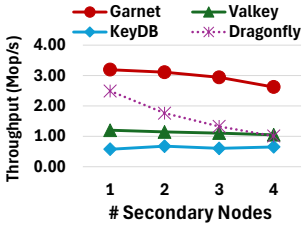
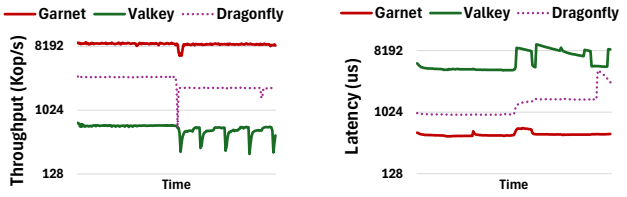


Figure 13: Write throughput with replication.



(a) Throughput. (b) P99.9 latency.

Figure 14: SET/GET throughput and latency during secondary synchronization.

write throughput. Dragonfly starts with higher throughput but quickly deteriorates with the addition of more secondaries.

Next, we evaluate how quickly the server can synchronize the attaching replicas in the presence of a heavy workload, and the impact of synchronization on throughput (Figure 14a)/latency (Figure 14b). For this experiment, we used 128 clients issuing a ratio 8(GET):2(SET) on a pre-warmed database that contains 256 million keys. While the workload is running, 4 replicas attach and synchronize with the primary. We see that Garnet achieves high throughput and is able to recover with minimal impact on the main workload. Dragonfly achieves lower throughput, although it is able to recover fast with minimal impact. However, its total throughput noticeably drops due to the increased overhead experienced by the primary having to serve the attached replicas. Valkey recovers quickly at the attach point but fails to keep up with the increased demand, resulting in frequent partial synchronization operations that affect throughput. We observed the same pattern for latency—Garnet manages to sustain the overall lowest P99.9 latency.

9 RELATED WORK

KV Stores and Caches. There is a rich body of related work on key-value stores and caches [9, 37]. FASTER [5, 24] is an embedded key-value library built for thread-scalable performance of point workloads. It follows the general model of a concurrent index over a record log. Compared to Tsavorite's `RUMDS` interface, FASTER's interface is not sufficiently expressive to handle the RESP API. It does not have multi-key transactions, read cache, revivification support, efficient compaction, streaming checkpoints, and cross-store checkpoint capability. Other embedded key-value stores include range-index-based stores such as RocksDB [4], LeanStore [30], and Bw-Tree [31]; however, these systems are designed to also support range queries, which are not necessary for a RESP cache-store, and thus incur the cost of reordering data, expensive compactions, page/level consolidation, and inner node traversals and binary searches during reads. Caches such as Kangaroo [38] and CacheLib [2] are designed as pure caches and may drop tuples, leading to an inability to be used as a cache-store. Further, all these systems target simple byte keys and values, and do not handle heap objects. As such, none of these systems meet the diverse requirements of cache-stores.

Cache-Stores. As discussed in Section 1, existing RESP cache-stores such as Redis, Valkey, Dragonfly, and KeyDB suffer from limitations in the areas of thread scalability, larger-than-memory support, checkpointing, fast scale-out, and tunable durability (Table 1 has more details). Shadowfax [26] adds a remote interface to FASTER, but is limited to simple string keys and values, similar to FaRM [11]. The Cassandra Query Language (CQL) is the cache-store interface used by Apache Cassandra [29] which is based on LSM-Tree [37, 40] storage. CQL is also adopted by ScyllaDB [54], which uses a shared-nothing design along with the Seastar [55] network routing framework. The overheads of a shared-nothing design have also been previously shown to result in 4× lower throughput (for memcached [39] with Seastar) at high thread counts [26].

Durability. A traditional consistent checkpoint requires the system to quiesce, which is not an option for a low-latency cache-store. One could take a fuzzy checkpoint [19, 34], but this would necessitate a write-ahead log for consistent recovery, which would impact scalability. Consistent checkpoints without the write-ahead log can allow a cache-store to remain scalable and allow recovery to a stale yet consistent copy, which is critical for warm cache scenarios. This led us to adapt the CPR [43] protocol for checkpoints, with changes as mentioned in Section 5.

10 CONCLUSIONS

Garnet is a new cache-store that fulfills key requirements that we recognize across a range of modern use cases: (1) Larger-than-memory, but memory-optimized (`LTM-MO`) capability; (2) Bare-metal performance; (3) Support for complex types and operations; and (4) Tunable durability. Garnet uses a hybrid shared-nothing session coupled with a shared-everything storage design. Garnet's storage engine, called Tsavorite, is a latch-free concurrent `LTM-MO` design with a narrow-waist interface that is used to express hundreds of commands. Tsavorite employs optimizations such as optimistic watching for transactions and epoch-based slot revivification for memory. Garnet has an efficient cluster design with sharding, replication, and migration protocols based on epoch-protected state machines. With tunable durability, Garnet can be deployed as either a cache or a database with varying data loss tolerance. Being Redis wire protocol compatible, it can serve as a drop-in replacement, offering up to orders-of-magnitude higher throughput and lower latencies than state-of-the-art RESP cache-stores. Garnet is widely used at Microsoft and is available in open-source.

ACKNOWLEDGMENTS

We would like to thank Johannes Gehrke, Surajit Chaudhuri, Phil Bernstein, Donald Kossmann, Knut Magne Risvik, and Shireesh Thota for their support, inspiration, and feedback. We also thank our product group partners, prior interns, and open-source contributors for helping improve the system and motivate with real use cases.

REFERENCES

[1] Jens Axboe. 2019. Efficient I/O with io_uring. https://kernel.dk/io_uring.pdf.

[2] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[3] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[4] Zhichao Cao and Siying Dong. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*.

[5] 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 *Proceedings of the 2018 International Conference on Management of Data*. 275–290.

[6] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[7] 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.

[8] National Research Council. 1994. *Realizing the Information Future: The Internet and Beyond*. The National Academies Press, Washington, DC. https://doi.org/10.17226/4755

[9] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2, Article 40 (April 2018), 43 pages. https://doi.org/10.1145/3158661

[10] Dell/Kioxia SSD Specs. 2025. https://aka.ms/AAx7d4d.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 401–414.

[12] Dragonfly. 2025. https://www.dragonflydb.io/.

[13] Dragonfly Architecture. 2025. https://aka.ms/AAx8vtf.

[14] Energy Sciences Network (ESnet). 2014. iperf3: A TCP, UDP, and SCTP Network Bandwidth Measurement Tool. https://software.es.net/iperf/.

[15] fio - Flexible I/O tester. 2025. https://fio.readthedocs.io/.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 51–64.

[17] Garnet - ETags. 2025. https://microsoft.github.io/garnet/blog/etags-when-and-how.

[18] Garnet API Compatibility. 2025. https://microsoft.github.io/garnet/docs/commands/api-compatibility.

[19] R B Hagmann. 1986. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Trans. Comput.* 35, 9 (Sept. 1986), 839–843. https://doi.org/10.1109/TC.1986.1676845

[20] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 683–692. https://doi.org/10.1145/2452376.2452456

[21] Stratos Idreos and Mark Callaghan. 2020. Key-Value Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2667–2672. https://doi.org/10.1145/3318464.3383133

[22] In-Memory Database Market Report. 2025. https://www.imarcgroup.com/in-memory-database-market.

[23] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[24] Konstantinos Kanellis, Badrish Chandramouli, and Shivaram Venkataraman. 2024. F2: Designing a Key-Value Store for Large Skewed Workloads. arXiv:2305.01516 [cs.DB] https://arxiv.org/abs/2305.01516

[25] KeyDB. 2025. https://docs.keydb.dev/.

[26] Chinmay Kulkarni, Badrish Chandramouli, and Ryan Stutsman. 2020. Achieving High Throughput and Elasticity in a Larger-than-Memory Store. In *PVLDB, 14(8), 2021*. https://www.microsoft.com/en-us/research/publication/achieving-high-throughput-and-elasticity-in-a-larger-than-memory-store/

[27] H. T. Kung and Philip L. Lehman. 1980. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. https://doi.org/10.1145/320613.320619

[28] Benjamin LaHaise. 2002. libaio: The Linux Asynchronous I/O Access Library. https://man7.org/linux/man-pages/man7/aio.7.html.

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[30] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 185–196.

[31] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (2013 ieee 29th international conference on data engineering (icde) ed.). IEEE. https://www.microsoft.com/en-us/research/publication/the-bw-tree-a-b-tree-for-new-hardware/

[32] LevelDB. 2025. https://github.com/google/leveldb.

[33] Tianyu Li, Badrish Chandramouli, and Samuel Madden. 2022. Performant Almost-Latch-Free Data Structures Using Epoch Protection. In *Proceedings of the 18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/3533737.3535091

[34] Jun-Lin Lin and Margaret H. Dunham. 1996. Segmented fuzzy checkpointing for main memory databases. In *Proceedings of the 1996 ACM Symposium on Applied Computing* (Philadelphia, Pennsylvania, USA) *(SAC '96)*. Association for Computing Machinery, New York, NY, USA, 158–165. https://doi.org/10.1145/331119.331168

[35] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[36] Lua Scripting Language. 2025. https://lua.org/.

[37] Chen Luo and Michael J. Carey. 2019. LSM-Based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (jul 2019), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[38] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. 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, 243–262. https://doi.org/10.1145/3477132.3483568

[39] Memcached. 2025. https://memcached.org/.

[40] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[41] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Rec.* 17, 3 (June 1988), 109–116. https://doi.org/10.1145/971701.50214

[42] W. W. Peterson and D. T. Brown. 1961. Cyclic Codes for Error Detection. *Proceedings of the IRE* 49, 1 (1961), 228–235. https://doi.org/10.1109/JRPROC.1961.287814

[43] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. 2019. Concurrent Prefix Recovery: Performing CPR on a Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 687–704. https://doi.org/10.1145/3299869.3300090

[44] Redis. 2025. https://redis.io/.

[45] Redis Cluster Specification. 2025. https://redis.io/docs/latest/operate/oss_and_stack/reference/cluster-spec/.

[46] Redis Enterprise Cluster Architecture. 2025. https://redis.io/technology/redis-enterprise-cluster-architecture/.

[47] Redis Labs User Survey (2016). 2025. https://redis.io/blog/redis-labs-customers-award-accolades/.

[48] Redis Lua API reference. 2025. https://redis.io/docs/latest/develop/interact/programmability/lua-api/.

[49] Redis Transactions. 2025. https://redis.io/docs/latest/develop/interact/transactions/.

[50] Redis Use Case Examples for Developers. 2025. https://redis.io/blog/5-industry-use-cases-for-redis-developers/.

[51] RedisLabs - Memtier Benchmark. 2025. https://github.com/RedisLabs/memtier_benchmark.

[52] RESP Serialization Protocol Specification. 2025. https://redis.io/docs/latest/develop/reference/protocol-spec/.

[53] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[54] ScyllaDB. 2025. https://scylladb.com/.

[55] SeaStar Framework. 2025. https://seastar.io/.
[56] Sliding Window Rate Limiter. 2025. https://redis.io/learn/develop/dotnet/aspnetcore/rate-limiting/sliding-window.
[57] 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* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 309–320. https://doi.org/10.1145/3626246.3653380
[58] The Resp.Benchhmark Tool. 2025. https://microsoft.github.io/garnet/docs/benchmarking/resp-bench.
[59] Transactions - Watch Command. 2025. https://microsoft.github.io/garnet/docs/dev/transactions.
[60] Valkey. 2025. https://valkey.io/.
[61] What is Memorystore. 2025. https://cloud.google.com/blog/topics/developers-practitioners/what-memorystore.