A Logically Disaggregated Cache for Replicated Storage Systems

Kiran Hombal
University of Illinois Urbana-Champaign

Henry Zhu
University of Illinois Urbana-Champaign

Shreesha G. Bhat
University of Illinois Urbana-Champaign

Neil Kaushikkar
Jump Trading Group

Ramnatthan Alagappan
University of Illinois Urbana-Champaign

Aishwarya Ganesan
University of Illinois Urbana-Champaign

Abstract

We study if replicated storage systems effectively utilize the caches embedded within each replica. Our study reveals that existing systems manage the embedded caches in each replica in silos, leading to significant *cache redundancy* across replicas and consequently low performance. To address this problem, we introduce *logically disaggregated cache* (Ldc), a new approach to managing caches in replicated storage systems. Ldc disaggregates the embedded caches from the replicas to form a single, logical cache. Ldc then allows any replica to access any part of the logical cache, which reduces redundancy caused by reads. Because writes pollute all caches, Ldc quickly demotes written objects to limit redundancy caused by writes. Ldc, however, realizes that reducing redundancy may hurt performance in some cases and thus employs an online analyzer to strike a balance between cache redundancy and coverage. We implement Ldc in three systems: an eventually-consistent KV store, a strongly-consistent KV store, and a production database. Using microbenchmarks, macrobenchmarks, and real-world traces, we show that the Ldc versions perform significantly better than the original systems (e.g., 2.6× to 5.4× higher throughput in the eventually-consistent KV store under YCSB).

1 Introduction

Replicated storage systems [15, 32, 55, 60, 63] are the backbone of modern datacenter applications. To reduce load on these storage systems and improve application performance, in-memory cache clusters (such as Memcached and Redis) are typically deployed in front of these services.

A key problem in these cache clusters, which contain many cache servers, is to efficiently utilize the in-memory cache space across the servers. This is an important problem because memory is an expensive resource, constituting up to 50% of a datacenter server's cost [8, 48]. Memcached-like systems address this problem by routing requests based on keys to different cache servers (for example, via hashing). Such routing makes the individual servers cache distinct objects, utilizing the cache space well. Hot objects are then replicated on a few cache servers to avoid load imbalance. Improving the space efficiency and performance of cache clusters has received significant attention from industry [7, 57] and academia [52, 59] alike.

While there has been a lot of focus on optimizing external cache clusters, scant attention has been paid to another set of caches: *caches that are embedded within the backend storage servers themselves.* Backend storage servers are replicated for fault tolerance, and each server stores the data on its disk and has an in-memory cache to speedup data accesses. For example, in MongoDB, each replica uses the WiredTiger local storage engine [56], and reads and writes at a replica happen via WiredTiger's cache [73]. If these embedded caches are not managed with care, each storage server could redundantly cache the same objects, which, in turn, could hurt performance as more requests must hit the disk.

This paper first studies how well existing replicated storage systems utilize their embedded caches by analyzing five popular systems: RethinkDB [60], Cassandra [15], RQLite [63], SplinterDB [70], and HBase [16]. Our overarching finding is that today's systems can *redundantly cache the same objects in the embedded caches across storage-server replicas, wasting precious cache space.* For example, in RethinkDB, caches across replicas can be highly similar, leading to as low as only 50% of the dataset being covered by the collective embedded caches, although it could have ideally fit the entire dataset. Cache redundancy in these systems is caused by both writes and reads. We find that writes cause redundancy because a write brings in the same object at all caches, as the write must be applied at all replicas to keep replicas consistent. Reads cause redundancy because different clients can read the same objects from different replicas over time.

Driven by this study, we seek a solution to reduce redundancy in embedded caches of replicated stores. At first sight, it may seem like key-based routing employed in external caches should solve the problem. However, we observe that routing is fundamentally inadequate for *embedded caches*

EuroSys '26, 27-30 April 2026, Edinburgh, UK

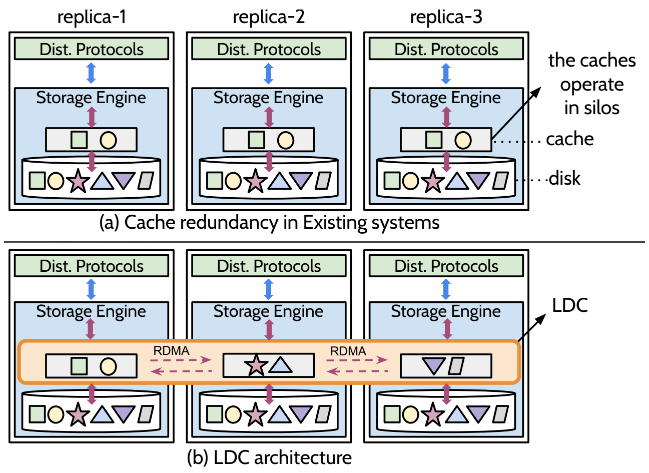


Figure 1. LDC overview. *(a) Shows how current systems suffer from redundant caching. (b) Shows LDC's proposed architecture where caches are logically disaggregated to form a single unified cache. Any replica can access any part of the logical cache via RDMA.*

for two reasons. First, while routing can avoid the redundancy caused by reads, it does not help with write-induced redundancy. External caches do not have to deal with write-induced cache redundancy because writes do not happen through the external cache; instead, writes update the backend storage and simply invalidate the object in the cache [57]. In contrast, a write at the backend storage system happens via the embedded caches at all storage replicas, which brings the written object into all the embedded caches. Second, even for reads, routing can help only when requests explicitly specify the keys (which can be used to route reads). While this is possible in external caches that expose a simpler key-value interface, backend storage systems often must handle *opaque* queries where the accessed objects are not explicitly specified in the request (e.g., nested queries, joins). Such queries cannot be correctly routed, making routing unviable to make the replicas cache distinct objects in their embedded caches.

This paper thus proposes a new architecture for managing embedded caches in replicated-storage backends called *logically disaggregated cache* (LDC). The core idea behind LDC is to disaggregate the embedded caches from the individual replicas to form a single, unified, logical cache. With LDC, the individual caches are aware of the fact that they are part of a logical cache and thus do not operate in silos. Figure 1 illustrates the LDC architecture.

The LDC architecture has two salient aspects that reduce cache redundancy. First, in LDC, any replica can access any part of the logical cache, even if the part physically resides on a remote replica. This design inherently reduces redundancy caused by reads: if an object is already present in the logical cache (within some replica's cache), the reading replica does not obliviously bring the same object from the disk into its cache, but instead directly fetches it from the remote cache. This limits redundancy and reduces disk accesses. Further, this design removes the need for correct query routing: any replica can handle a query, including opaque ones, and fetch the required objects from any part of the logical cache. LDC's design leverages one-sided RDMA operations to access remote parts of the logical cache at low latencies and without CPU interruption[†].

Second, unlike in existing systems, the individual physical caches in LDC are aware that they are part of a logical cache and that a write will cause the same object to be redundantly cached everywhere. The caches thus handle writes in a way that limits redundancy. A naive way to achieve this is to have all replicas bypass their cache and write directly to disk, but this significantly slows down writes. Instead, LDC allows writes to happen via the caches but *quickly demotes* written objects to limit cache redundancy. LDC does such quick write demotions *selectively* on some replicas. The idea is that if a replica freshly brings an object into its cache for writing, it would increase the redundancy and so the replica quickly demotes the object. In contrast, if a replica already has the object in its cache (e.g., from a previous read), it would not add to the redundancy, and so the replica does not quickly demote the object. LDC's design uses a separate tiny queue (apart from the main cache) to realize selective quick write demotions with ease. Overall, the above two aspects enable LDC minimize cache redundancy from both reads and writes.

The LDC design described thus far prioritizes reducing cache redundancy to improve coverage. However, when some objects are more popular than others, LDC can incur repeated RDMA accesses to those objects; while fast, RDMA is still slower than local memory accesses. Thus, caching such objects at multiple replicas, even if it increases redundancy, can be beneficial. However, in some situations, redundantly caching even popular objects can deteriorate performance: a redundant object displaces one or more cached objects, triggering future disk accesses. Thus, the system must balance coverage and redundancy. LDC does so using an *online cost-benefit analyzer*. At a high level, the analyzer only admits remote objects for which the benefits due to redundantly caching them outweigh the cost of reduced coverage.

Implementing LDC in existing replicated storage systems does not require intrusive changes to distributed protocols or the storage engine; all changes can be contained within the existing cache layer. The rest of the system continues to interact with the cache layer as usual. This ease of integration has enabled us to implement LDC in three systems with moderate effort: TWIG-KV, a home-grown, eventually-consistent key-value store; CRAQ-KV, a variant of TWIG-KV that uses the CRAQ replication protocol [68] to realize linearizability; finally, to show that LDC can be applied to production system, we built a LDC version of RethinkDB.

Our experiments show that the LDC version of TWIG-KV

[†]While our implementation uses RDMA for its benefits [40, 41], the LDC idea can be realized using RPCs as well.

greatly reduces cache redundancy when coverage is important and correctly caches popular object on many replicas under skewed workloads, thus improving throughput by up to 5.3× and 4.5× in read-only and read-write workloads, respectively. We run extensive microbenchmarks with different distributions, cache sizes, and number of replicas, and show that Ldc offers benefits in all cases. We show that Ldc offers higher performance than routing under read-write workloads and opaque queries. We then compare Ldc to two approaches that can possibly reduce cache redundancy: cooperative caching [29] where entities cooperatively cache data, and better cache policies at each replica that utilize the cache space well. Ldc has performance advantages over both these schemes. Under YCSB, Ldc improves the performance of Twig-kv and Craq-kv by 2.6×–5.4× and 2.5×–5.9×, respectively. In RethinkDB, Ldc improves read performance by 1.3× to 1.9×. Finally, with 50 real-world traces (from Twitter, Meta, and Alibaba), we show that Ldc balances redundancy and coverage well, reducing latency on average by 1.57× and up to 5.76×.

In summary, this paper makes the following contributions:

- We first bring to light the problem of redundant caching in the embedded caches of replicated key-value stores and show this problem in five existing, popular systems.
- We propose logically disaggregated cache, a novel architecture to manage embedded caches across storage replicas, which improves cache efficiency and thus performance.
- We implement the Ldc design in three systems and experimentally demonstrate their benefits.

2 Analyzing the Efficacy of Embedded Caches

We first provide a brief background on replicated stores, and how they process reads and writes. We then present our analysis of how current systems utilize the embedded caches and argue why existing approaches in external caches are insufficient to address this problem.

2.1 Background

Distributed storage systems store data on many replicas for fault-tolerance, so that even if a few replicas fail, the data can continue to remain available[†]. In such a replicated system, each replica internally has two layers: a distributed-protocol layer that helps coordinate actions across replicas (e.g., to replicate updates); a storage engine that is responsible for managing data that resides on disk. The storage engine usually organizes on-disk data using an index (e.g., a B-tree). The storage engine on each replica uses an embedded in-memory cache. While some engines cache at the granularity of objects, others do so at page granularity (e.g., B-tree pages).

[†]While some cloud-native databases offload fault-tolerance to cloud storage services, these cloud services ultimately replicate data across servers for fault-tolerance. Our focus is on such shared-nothing replicated stores.

System	Replication protocol	Local storage engine	Caching granularity	Writes via cache?	Reads via cache?
RethinkDB [60]	Raft	custom B-tree	Page	√	√
RQLite [63]	Raft	SQLite (B-tree)	Page	√	√
Replicated SplinterDB [70]	Raft	SplinterDB (B^{ϵ}-tree)	Page	√	√
HBase [16]	Read replicas	SSTable	Block	×	√
Cassandra [15]	Quorums	SSTable	Object (row)	×*	√

* – writes go through the OS page cache.

Table 1. Characteristics of Analyzed Systems.

Most practical systems provide eventual consistency for reads and thus allow reads at any replica [3, 4, 22, 32]. Some systems offer strong consistency while allowing reads at any replica via additional mechanisms [19, 68, 76]. Irrespective of the read protocol implemented by the system, the replica receiving the read invokes the storage engine, which, in turn, accesses the required objects via its embedded cache.

Writes must be applied at all replicas to keep data consistent. Typically, writes are done via a designated leader (or the coordinator in quorum systems [32]). The leader replicates a write, applies it, and acknowledges the client. Replication can be synchronous or lazy. If synchronous, a write is replicated to all or a quorum before acknowledgment [37, 51, 58, 61]. If lazy, writes are replicated asynchronously. Regardless, all replicas must apply writes (at least eventually) and each replica does so via its embedded in-memory cache. Specifically, the object is first dirtied in the cache and periodically, dirty objects are written back to disk.

2.2 Analysis of Current Systems

We analyze if existing replicated stores effectively utilize the collective embedded-cache space. We do so by quantifying the redundancy across caches using a metric called *cache similarity factor* (*CSF*). CSF is given by $(1 - (|\bigcup_{i=1}^{N} C_i| / \sum_{i=1}^{N} S_i)) * (N / (N-1))$, where N, S_i, and C_i are the number of replicas, cache size at replica i (in terms of the number of objects it can fit), and the set of objects in the cache of replica i, respectively. CSF is based on a set-similarity metric involving more than two sets [33]. CSF ranges from 0 to 1. As an example, with $N = 3$ and $S_i = 10$, if all caches hold the same 10 objects (i.e., $|\bigcup_{i=1}^{N} C_i| = 10$), then CSF = 1. If all caches hold completely different objects ($|\bigcup_{i=1}^{N} C_i| = 30$), then CSF = 0. Intuitively, higher CSF means that the caches contain many redundant objects and thus less of the dataset resides in memory (i.e., cache coverage is low). On the contrary, lower CSF indicates that the caches are more exclusive and thus a larger fraction of dataset is covered by the cache. As we show (§5), lower CSF can lead to higher performance under many workloads.

Setup. Table 1 shows the systems we analyzed. These systems use different replication protocols, storage engines, and

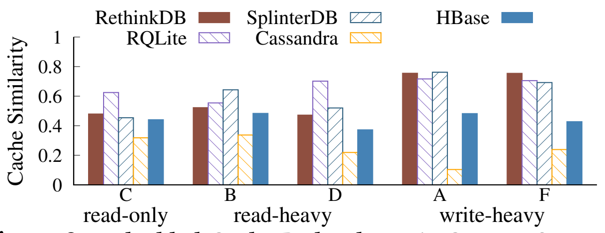


Figure 2. Embedded-Cache Redundancy in Current Systems.

caching granularities. All systems allow reads at any replica. The Cassandra client issues operations to different replicas in a round-robin manner [31]. This is Cassandra's default load-balancing policy, which distributes reads across replicas; this increases cache redundancy compared to sticky sessions, as different replicas serve reads for the same objects. In others, the client maintains a sticky session with one replica for reads and another session with the leader for issuing writes. We measure CSF with three replicas with the YCSB [28] benchmark which has read-only (C), read-heavy (B, D), and write-heavy (A, F) workloads. We use 24-byte keys and 1KB values (but our results hold for other sizes as well). We configure the cache on each replica to be 33.3% of the dataset size; we choose this size because an ideal system can collectively cache the entire dataset across the three replicas. Figure 2 shows the results.

RethinkDB. RethinkDB replicates updates via Raft [58], uses a B-tree storage engine, and caches the tree pages [62]; both writes and reads happen via the cache. First, under the read-only workload, the caches are highly similar (~50%) because different clients read the same objects at different replicas. RethinkDB caches at the granularity of pages (which internally contains multiple objects), bringing objects into the cache that are not necessarily read; this also contributes to the CSF. Second, writes bring in the same objects at all caches as the leader replicates a write to all replicas. Thus, under read-heavy workloads, CSF slightly increases because the few writes bring in the same pages into all caches. Under write-heavy workloads, CSF-s increase significantly (to 75%) as more operations bring in the same pages across caches. Measuring the cache coverage shows that such high redundancy leads to only about 50% of the dataset being cached, while ideally the replicas could have collectively cached 100% of the dataset.

RQLite. RQLite [63] is a distributed database that uses Raft for replication and SQLite [6] as the storage engine. SQLite internally caches B-tree pages [5] and all reads and writes happen via the cache. Similar to RethinkDB, RQLite has high CSF, with write-heavy workloads having higher CSF values.

Replicated SplinterDB. SplinterDB is a production single-node database used within VMware [70]; we built a replicated version by overlaying a popular Raft library [35] atop the SplinterDB storage engine. SplinterDB uses a B$^{\epsilon}$-tree [27] and caches the tree pages, and reads and writes happen via the cache. Its results are similar to that of RethinkDB.

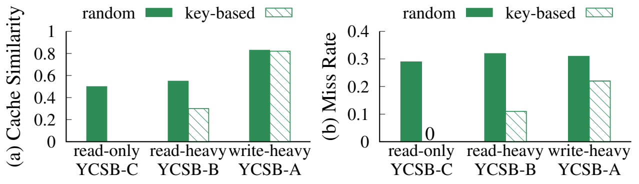


Figure 3. Key-Based Routing with Writes.

Cassandra. Cassandra uses quorum replication [32, 46] and SSTable format [23] for storage and it caches rows in a user-level cache [30]. Reads go through the row cache, while writes happen via the OS page cache and they invalidate the row cache. Since writes do not pollute the row cache, write-heavy workloads do not have higher CSF-s than read-heavy workloads (unlike RethinkDB and SplinterDB) as we measure only the row cache. However, read-only and read-heavy workloads still have high CSF-s. Cassandra's CSF values are generally lower than the previous systems because Cassandra caches at the granularity of row, which avoids caching rows that are not read. Under write-heavy workloads, CSF reduces significantly because a write invalidates the row in all caches.

HBase. HBase uses read-only replicas [1] in addition to the main region server [2] for improved read throughput. The replicas cache data from underlying HDFS which stores the data in the SSTable format. Caching happens at the SSTable block granularity. HBase has slightly higher CSF than Cassandra in general because of its block-granular caching. Writes in HBase do not happen via the replicas' caches; thus, CSF in write-heavy workloads does not increase much compared to read-heavy workloads. Since HBase does not invalidate cached data upon writes, under write-heavy workloads, it has a higher CSF than Cassandra.

Summary. Today's replicated stores exhibit high cache redundancy. An ideal system must reduce cache redundancy and effectively utilize the collective embedded cache space. This increases cache coverage, which is critical for performance under many workloads. For example, uniform workloads that access a large fraction of the dataset benefit from higher cache coverage. Even skewed workloads (e.g., zipfian) benefit from higher coverage, as our experiments show (§5).

2.3 Why Don't Techniques from External Caches Work

External caches reduce redundancy across cache servers by routing based on keys [52, 59]. While this works for external caches, unfortunately, it does not suffice for embedded caches as we explain below.

Does Not Handle Writes. First, routing cannot simply handle cache redundancy caused by writes. In external cache clusters, such as Memcached, writes happen at the backend storage system and the copies in the cache servers are simply invalidated [57]. In contrast, writes at the backend replicated storage system happen via the embedded caches at all replicas, polluting the caches. To show the extent to which writes cause problems, we adopt the key-based routing scheme to embedded caches, where reads are routed to replicas based on the key. We also compare this approach to a random-routing scheme, where clients send reads to a random replica. In both these schemes, writes are applied at all replicas through the caches. As shown in Figure 3, in the read-only workload, key-based routing achieves zero CSF and misses, while random has high CSF and miss rates, showing the effectiveness of key-based routing. However, when the workload contains just 5% writes (YCSB-B), key-based routing starts to suffer from cache redundancy and more misses as writes start polluting all caches. With a 50% write workload (YCSB-A), the problem worsens, with key-based routing having a CSF as high as random routing (although routing based on keys helps a little with miss rates compared to random routing).

Needs Accurate Routing. Another problem with key-based routing is that it requires accurate routing for even reads to utilize the caches well. This works well in external caches with a simple key-value interface, where the read keys are explicitly specified in the request. However, in backend replicated storage systems, accurate routing is hard because requests do not always explicitly specify the objects. For example, it is hard to route nested queries using key-based routing. With a nested query, the inner query must be first executed to find which rows will be accessed in the outer query, making routing difficult. One possible way is to push the query processing to the client, which can first execute the inner query to find the rows to access for the outer query. The client can then get every row in the outer query. However, this approach is impractical as it requires clients to do complex query processing, increases roundtrips, and transfers more data between the client and the servers. Another option is to send such queries to a random replica, which can execute it and read the data from disk if it is not in its cache. However, this does not utilize the embedded caches across servers well and increases disk accesses. Similarly, it is difficult to do routing with join queries.

Overall, the techniques used by external caches to ensure cache efficiency are inadequate for embedded caches in replicated stores. We later show how LDC avoids the above problems and improves performance over routing (§5.6).

3 The LDC Architecture

3.1 Main Ideas and Overview

Logically disaggregated cache (LDC) is a new architecture to manage embedded caches in replicated storage systems. LDC's main idea is to disaggregate the embedded caches from the individual replicas to form a single, unified, logical cache. With LDC, the individual embedded caches are aware of the fact that they are part of a logical cache and thus do not operate in a silo. Figure 4 shows the high-level architecture of LDC. The logical cache (the orange box) is internally composed of the individual physical caches local to the replicas.

LDC has two critical aspects that depart from how embedded caches are managed today. First, in LDC, any replica can access any part of the logical cache, irrespective of whether the part belongs to the local replica or a remote replica. This design naturally reduces redundancy caused by reads: if an object is already present within some replica's cache, any replica can directly fetch it (instead of admitting the object into its cache). This not only reduces cache redundancy but also improves performance by reducing disk accesses. This design also obviates the need for correct request routing; clients can send requests to any replica, which eases handling opaque queries. Take the nested-query example from §2.3. With LDC, the receiving replica would process the inner query, accessing the relevant rows from either its local cache or remote caches, which would enable the replica to know which rows must be accessed for the outer query. The replica then would fetch those rows from the local or remote caches. §3.2 describes how LDC realizes efficient and correct remote accesses via one-sided RDMA.

Second, unlike existing systems, the individual physical caches in LDC are aware that they are part of a single logical cache and thus handle writes carefully, avoiding bringing the same object into all physical caches. A naive way to realize this would be to bypass the caches and write directly to storage; however, this approach will significantly impact write performance as writes cannot be coalesced or buffered. Instead, LDC uses an approach that we call selective and quick write demotions. LDC still allows writes to happen via the caches, but since the individual physical caches are aware that writes will increase cache redundancy, they *quickly demote* written objects. LDC does such quick write demotions *selectively* only on some replicas. If an object is freshly brought into the cache at a replica just to perform the write, then cache redundancy will increase considerably. Thus, LDC would quickly demote the object on that replica, keeping the redundancy in check. However, if the object being written is already present in the cache at a replica (e.g., due to a previous read), it will not add to the redundancy, and so LDC does not quickly demote the object on that replica. Such selective and quick demotion helps LDC tame write-induced redundancy. §3.3 describes how LDC realizes selective and quick demotion using a separate tiny queue.

While reduced redundancy (equivalently, increased coverage) improves performance under many cases, in some cases, it can also hurt performance. For example, under skewed workloads, simply reducing redundancy might trigger repeated remote cache accesses. Under such workloads, redundantly caching the popular items is more beneficial. Thus,

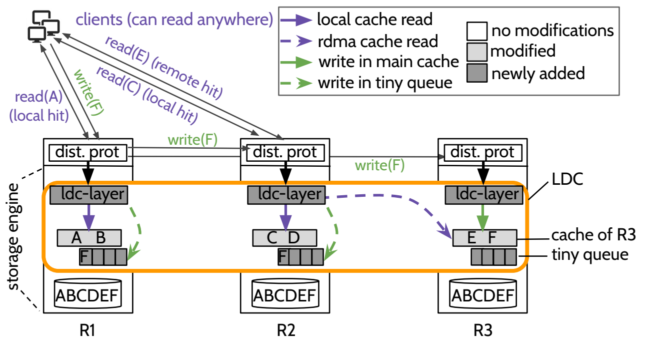


Figure 4. Ldc. *The figure shows LDC's architecture and the read (purple arrows) and write paths (green arrows). R1-R3: the three storage replicas. A-F: objects stored by the system in this example. Implementing LDC requires no changes to distributed protocols or storage layer but only the cache (§4).*

the system must balance cache coverage and redundancy to realize high performance. §3.4 describes how Ldc achieves this using an online cost-benefit analyzer.

3.2 Efficient and Correct Remote Cache Access

When a replica accesses a part of the logical cache that physically resides on the same replica, the access is local. If not, the access is remote. Remote accesses must satisfy two requirements. First, they must not increase the request load on the remote replica. Second, remote accesses must be fast.

Ldc leverages one-sided RDMA to satisfy the above two requirements. First, RDMA accesses are "CPU-free": a remote replica's cache can be accessed without CPU interruption; thus, remote cache accesses do not increase the remote replica's CPU load. Second, RDMA offers low latencies (a few μs). Thus, it offers a faster option over accessing storage which incurs 100s of μs with SATA SSDs [75] and can be even quicker than accessing NVMe SSDs which incur 10s of μs [75]. Ldc's design is viable given the ubiquity of RDMA in data centers [40, 41, 82]. Even when replicas are across fault domains (requiring traversal of many switches), RDMA accesses are faster than going to the disk in modern environments [82]. RDMA is not fundamental to Ldc; it can also be realized with RPCs (albeit with CPU processing at the receiver side and slightly higher latencies).

While one-sided RDMA avoids CPU interruption at the receiver end, the replica initiating the remote access must know the location of the object within a remote cache to fetch it directly over RDMA. To do this, in Ldc, each replica locally maintains an index for remote caches. Fortunately, in many systems, each replica already has an index over its local cache to locate objects (e.g., a hashmap that maps pages to addresses in the local cache). In Ldc, the replicas exchange these cache indexes; thus, Ldc can lookup the exchanged indexes to know which objects are present in the remote caches and their locations within the cache. When a replica updates its local cache index (e.g., upon an eviction), Ldc exchanges index updates with other replicas. To reduce overhead, Ldc propagates index updates to other replicas only lazily.

However, one problem with lazy propagation is that the copy of index for a remote cache can become stale. Thus, it is possible for two replicas to find that an object does not exist in any remote cache and thus page in the same object into their local caches. Ldc does not actively prevent this situation because this happens infrequently. For example, under a zipfian workload, only 0.5% of the remote accesses result in a wrong lookup because of a stale index. This results in a fallback to disk reads. In general, Ldc is not designed to eliminate redundancy but rather it takes a best-effort approach to minimizing redundancy.

Another challenge with lazily exchanging indexes is that Ldc may fetch a different object than the one intended from the remote cache (say, because a different object was paged into that location). Ldc solves this by treating the index of a remote cache as a hint and checks if a remote read actually returned the intended object. To do this check, Ldc stores the object identifier as part of the cached object. If the check fails, Ldc falls back to local disk, reading the object from disk into the cache. Another challenge that arises with RDMA is that an object being remotely read could be concurrently written to locally at the target cache; thus, a remote read may see partial writes. To handle this, Ldc augments the cache objects with a checksum, which is verified upon a remote read, retrying the read if the check fails.

Figure 4 shows how reads work in Ldc. Clients can route reads anywhere. If the object is present in the local cache of the replica handling the read request, the read is served locally. This is a *local hit* (e.g., read(A) at replica R1). If the object is not locally cached, Ldc checks the local copy of the remote cache indexes to see if the object is present in some other remote cache; if so, it transparently fetches it over RDMA. This is a *remote hit* (e.g., read(E) at R2, which fetches E from R3's cache). Given that RDMA accesses are fast, remotely fetched objects are not admitted into local caches by default. This helps maintain the cache redundancy low. If the object is not cached anywhere (i.e., a miss), then Ldc brings in the object from its disk into its local cache. This updates the cache index on that replica. Once exchanged, the object is now available for remote accesses.

3.3 Selective and Quick Write Demotions

Ldc's idea to reduce write-induced redundancy is to quickly demote written objects that are not already in cache. Ldc realizes quick demotions by maintaining a *separate tiny queue* in addition to the main cache at each replica. The queue is private to each replica (other replicas cannot access it) and the queue is carved from the main cache without increasing the total cache size. If the object being written is not already present in the main cache at a replica, Ldc admits the object into the queue. If not, Ldc updates the object directly in the main cache. Objects in the queue are evicted and written to disk periodically. The tiny queue is carved from the main cache; it is typically a very small fraction for example 0.01% of the cache.

The small queue helps minimize cache redundancy while not compromising write performance. In particular, writing via the tiny queue restricts the redundancy induced by writes to that of the size of the queue instead of polluting the main cache. At the same time, the small queue helps avoid direct disk writes. Of course, compared to writing via the main cache, the tiny queue reduces write performance as writes may have to wait for tiny-queue evictions. However, selectively doing quick demotions (only on replicas where the object is freshly brought in for writes) minimizes this impact. Specifically, writes to popular objects will be absorbed in the main cache because LDC would redundantly cache such objects on many replicas (as we describe in the next subsection). Besides, the slightly reduced write performance is a small cost to pay for the improved cache coverage, using which reads can significantly be sped up in many workloads.

If an object in the tiny queue is read, LDC promotes the object to the main cache at that replica but it does so without increasing redundancy. Specifically, LDC promotes the object into the replica's main cache only if no other replica has already done so. This is because, if it is available in the main cache of another replica already, then it can be served directly from there instead of polluting the local (main) cache. Figure 4 shows the write path in LDC. When an update to object F is applied at the replicas, replica R3 has F in its main cache and thus updates the object in the main cache. The other replicas do not have F in their main caches and thus the object is added to the tiny queues for quick demotion.

3.4 Balancing Coverage and Redundancy

The LDC design described so far reduces redundancy introduced by both reads and writes. Such reduced redundancy improves cache coverage, which is critical for performance. However, when a few objects are very popular, repeatedly accessing them over RDMA accesses can be expensive; instead, it may be beneficial to redundantly cache such objects on many or all replicas. Also, redundantly caching popular objects helps LDC absorb writes in the main cache on many replicas. However, under certain situations, even when objects are repeatedly accessed over RDMA, it might be beneficial to *not* redundantly cache them. For example, consider a workload where almost all accesses are to 30% objects and each of the three replicas has cache space which is 10% of the dataset. Here, it is beneficial to optimize for coverage, collectively containing all hot objects in the cache. Thus, an ideal solution must balance between coverage and redundancy by considering the workload and the cache size.

Intuitively, there is a cost and a benefit to redundantly caching an object O. The cost is that O would displace one or more objects from the cache (more because O may be cached at many replicas), which reduces cache coverage, triggering disk accesses for the displaced objects. The benefit

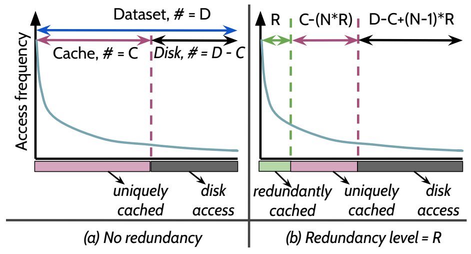


Figure 5. Cost-Benefit Analysis. *The figure shows the analyses performed by the CBA. In (a), the CBA explores the point with no redundancy. In (b), the CBA explores the impact of redundantly caching top R objects on all N replicas, which decreases cache coverage by $(N-1)*R$.*

is that RDMA accesses to O can be converted into local accesses. One must choose to redundantly cache O if the potential benefit outweighs the cost. Thus, LDC implements a *cost-benefit analyzer* (CBA) that balances coverage and redundancy. At a high-level, the CBA determines the optimal level of redundancy (R_{opt}) by estimating performance for different redundancy levels. Then, upon a remote read, the CBA checks if the accessed object is among the R_{opt} most frequently accessed objects and, if so, locally admits the object. If not, the object is served without local admission.

We now explain how CBA estimates the performance for a redundancy level R, which we denote as $P_{est}(R)$. R_{opt} is the value of R that maximizes P_{est}. The CBA runs periodically (every t seconds) to determine R_{opt}. At a high level, the CBA has knowledge of access frequencies of objects accessed in the current window (i.e., the last t seconds). Let A be the access frequencies, sorted in decreasing order. C is the total size of the unified cache (i.e., caches across all N replicas); D is the dataset size. When there is no redundancy, $A[1..C]$ correspond to the access rates of objects in cache (assuming a caching policy optimally caches top C objects); $A[C+1..D]$ correspond to the ones on disk. Figure 5(a) shows this case.

Intuitively, if we increase the redundancy level to R, then the R most frequently accessed objects will become redundantly cached on all N replicas. This reduces the cache coverage by $(N-1)*R$, leading to $(N-1)*R$ additional objects being accessed from disk. Thus, for a given redundancy level R, the cache will have $C-(N-1)*R$ objects with R of those being redundantly cached. The remaining $D-C+(N-1)*R$ objects will be accessed from disk. This is shown in Figure 5(b). For redundantly cached objects, their accesses will be local. For objects that are cached by only one replica, roughly $\frac{1}{N}$ of requests will be local and the remaining $\frac{N-1}{N}$ will be over RDMA; this is because a cached object at one replica can be remotely accessed by any of the remaining $N-1$ replicas. Let l_{local}, l_{rdma}, and l_{disk} be the access latencies for local cache, remote cache over RDMA, and disk, respectively. Thus, the estimated latency for a given redundancy level R is given by $L_{est}(R) = L_{redundant}(R) + L_{unique}(R) + L_{disk}(R)$, where

$$L_{redundant}(R) = \sum_{i=1}^{R} A[i] * l_{local}$$

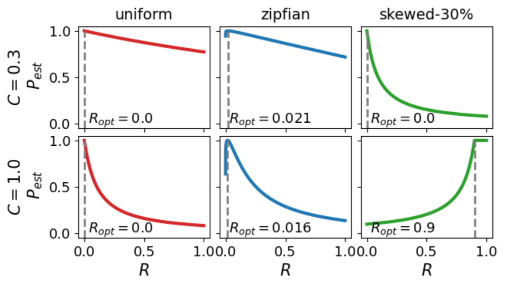


Figure 6. *R* **Values vs. Estimated Performance.** *The figure plots how P_{est} varies with redundancy (R) for different workloads (top: total cache size is 30% of dataset, bottom: 100%). The dashed line shows the R_{opt} point. Note that the x-axis is normalized with respect to per-replica cache size.*

$$L_{unique}(R) = \sum_{j=R+1}^{C-(N-1)*R} A[j] * \left(\frac{l_{local}}{N} + \frac{(N-1)*l_{rdma}}{N}\right)$$

$$L_{disk}(R) = \sum_{k=C-(N-1)*R+1}^{D} A[k] * l_{disk}$$

The estimated performance $P_{est}(R)$ is simply $1/L_{est}(R)$. The CBA then finds the optimal redundancy level (R_{opt}) that maximizes $P_{est}(R)$. Whenever a replica accesses an object over RDMA, it checks if the object is among the R_{opt} most frequently accessed objects in A. If so, the replica admits the object into the local cache. If not, it serves the read request without admitting the object into its local cache. This policy thus achieves the redundancy level of R_{opt}. If the CBA decides to reduce R_{opt} in a subsequent iteration, then the replica stops admitting any new objects until the redundancy level falls below R_{opt} due to evictions. If a replica fails, the CBA adjusts the capacity and determines the new R_{opt}. In our current implementation, every replica runs the CBA independently and determines R_{opt}. Our implementation uses measured access latencies in the previous window for l_{local}, l_{rdma}, and l_{disk} (instead of using constants), thus taking into account the current network and disk contention.

Figure 6 shows how the CBA picks R_{opt} when executing three different real workloads with three replicas. We show two cache sizes: 10% at each replica ($C = 0.3$, top row in the figure) and 33.3% at each replica ($C = 1.0$, bottom row). For uniform workloads, zero cache redundancy leads to higher performance because all keys are uniformly likely to be accessed and thus increasing coverage is important; the dashed line shows that the CBA picks this point as R_{opt} for both cache sizes. For zipfian, a few objects are more popular than others. Thus, redundantly caching those objects offers the highest performance. However, redundancy beyond a point reduces performance because coverage becomes low. The CBA chooses the peak as R_{opt}. In the skewed-30% workload, most accesses go to 30% of objects. Here, with each replica having 33.3% cache ($C = 1.0$), it is optimal to redundantly cache the 30% objects everywhere. In contrast, with $C = 0.3$, it is best to optimize for coverage and collectively utilize the space to cache all 30% objects. The CBA correctly chooses these points based on the cache size.

3.5 Fault Tolerance and Availability

LDC introduces no additional single points of failure: it provides the same fault-tolerance properties as the baseline system (e.g., it can tolerate replica crashes and network partitions). The only additional failure handling that must happen during reads is that, when a replica reads from a remote replica that has crashed, RDMA accesses will fail. LDC handles this by falling back to the disk-access path. In such situations, performance is no different from the baselines, which would also read from disk if the object is a local cache miss. Furthermore, CBA accounts for replica failures by adjusting the total cache capacity C and recomputing R_{opt}, ensuring that LDC continues to balance coverage and redundancy appropriately with the remaining replicas.

4 Integration into Existing Systems

LDC can be implemented in an existing system by making changes only to its cache layer, without requiring changes to the distributed protocols or the storage engine of the original system. The rest of the system interacts with LDC the same way as with its regular cache. Internally, LDC manages the local caches as a unified cache and fetches objects from the local or remote caches transparent to the layers above. Two logical changes are required to realize LDC (as shown in Figure 4): a set of modifications to the original local cache and a new LDC layer that sits on top of the local cache.

We implement LDC in two key-value stores (TWIG-KV and CRAQ-KV) and a production database (RethinkDB). TWIG-KV and CRAQ-KV are research prototypes that allow us to test different LDC design variants, for example LDC without selective and quick write demotions. RethinkDB is a production system with a large, mature codebase. Integrating LDC into it validates that our design can be adopted in real-world systems with minimal effort. Together, these systems demonstrate the flexibility and practicality of LDC's design.

Changes to Local Cache. Although LDC requires changes to the local cache, it can continue to use the original cache's data structures, granularity (e.g., pages or objects), and eviction policy. The changes required are for interfacing with the LDC layer. The first change is to set up the cache as an RDMA-able region with the correct protections (read permission for all replicas and read-write for the local replica). Second, the cache must expose its index and share index updates to the LDC layer. Third, the cache augments the cached objects with object identifiers and checksums to enable correctness checks for remote accesses (§3.2). Fourth, the cache must expose a way for LDC layer to admit objects (e.g., from the tiny queue if not cached elsewhere or remote cache when CBA decides so). Finally, the cache must expose a way for LDC layer to control whether or not an object read from disk can be admitted into the main cache. This is needed because when paging for write, the object should not be admitted into the main cache but the tiny queue. Upon a read, however, Ldc would ask to admit to main cache (to make the object available for others).

Ldc Layer. The Ldc layer sits on top of the local cache. It manages RDMA connections, performs checks over remotely accessed data, exchanges cache-index updates, and runs the CBA. The Ldc layer makes various admission decisions: whether or not to admit objects into main cache from tiny queue or remote cache and whether or not to admit to main cache when paging in from disk. It also allocates and manages the tiny queue, admitting an object into the queue for a write if the object is not already in the main cache.

To build the Ldc version of a system, only moderate effort is required. The local-cache changes are usually straightforward and most parts of the Ldc layer (e.g., CBA) can be reused across systems; only parts of Ldc that interact with the local cache need to be implemented for each system. As evidence, we have implemented Ldc in three systems.

4.1 Twig-kv: An Eventually Consistent Store

Twig-kv is a home-grown, lightweight, eventually consistent key-value store that uses primary-backup replication [20]. Writes are sent to the primary (leader) which asynchronously replicates to the backups (followers). Reads can be served by any replica. The storage layer stores key-value pairs (along with a checksum for integrity) on disk and uses an index for fast lookups; it caches key-value pairs and all writes and reads at a replica happen via the cache. The cache uses LRU eviction and an index to lookup keys within the cache.

The Ldc version changes the cache layer as above but preserves the cache data structures, granularity, and policies. The Ldc layer implements the functionality mentioned above. It reuses the checksums used by the original version and keys which are stored along with the values in the cache to validate remote reads. Our implementation modifies 210 LOC in the cache and adds 2.5KLOC in the Ldc layer.

4.2 Craq-kv: A Strongly Consistent Store

Craq-kv is a variant of Twig-kv that uses the CRAQ distributed protocol [68] for replicating writes and realizing linearizable reads. In CRAQ, replicas are arranged as a chain. A write starts at the head of the chain and traverses through each replica until it reaches the tail after which it is considered committed. Upon a write, a replica increments the *dirty version* number of the object and propagates the write down the chain. When the write is committed, the tail back-propagates a commit, upon which each replica updates the object's *clean version*. A read at a replica is locally served if the object's clean and dirty versions match. Otherwise, the replica fetches the object from the tail replica and serves it, which ensures strong consistency. Craq-kv's storage layer is almost the same as Twig-kv's but it also stores and caches the dirty and clean version numbers for every object.

In the Ldc version, a replica can fetch an object from a remote cache if it is not in the local cache. To maintain strong consistency during remote cache reads, Ldc also fetches the clean and dirty versions of the remote object and returns it to the distributed layer, which then performs the version checks as above. Thus, remote cache read does not alter the core consistency logic of CRAQ: it is as if the read were received by the remote replica and that replica served it from its local cache. The code changes required for the Ldc version of Craq-kv are similar to that of Twig-kv.

4.3 RethinkDB: A Production Database

RethinkDB is a production database. As discussed in §2.2, RethinkDB uses Raft for replicating writes. It allows reads at any replica including the followers (albeit with no consistency guarantees). RethinkDB uses a custom B-tree as its storage engine. The storage engine caches B-tree pages and all reads and writes at a replica happen via the cache.

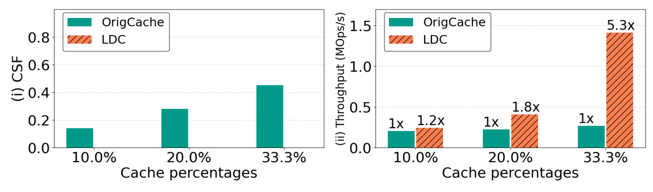
In our current Ldc version of RethinkDB, we focused on the read-path optimizations and did not implement the write path due to time constraints. However, it is fairly straightforward to include it in the future. We changed the allocation of the local cache to be contiguous so that it can be set up as an RDMA region. All modifications are contained within the cache layer, with no changes to RethinkDB's distributed protocols or storage engine internals. We add/change 2.4KLOC: 850 lines to implement local cache changes (including allocation) and 1.5KLOC for common Ldc functionality. Overall, we modify <1% of RethinkDB's 282KLOC codebase, a small effort for Ldc's performance gains (§5.10).

5 Evaluation

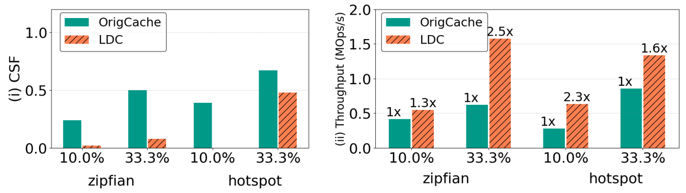
In our evaluation, we answer the following questions:

- Does Ldc offer benefits under read-only workloads? (§5.1)
- Does Ldc offer benefits under read-write workloads? (§5.2)
- How does Ldc's benefits scale with replicas? (§5.3.1)
- At what cache size does the unmodified system match the performance of the Ldc version? (§5.3.2)
- Does Ldc remain beneficial with fast NVMe storage? (§5.4)
- Does Ldc scale with larger datasets? (§5.5)
- How does Ldc compare to request routing? (§5.6)
- How does Ldc compare to related approaches like cooperative caching and better replica-local cache policies (§5.7)?
- How does Ldc help improve Twig-kv's performance under the YCSB macrobenchmark? (§5.8)
- Does Ldc benefit strongly consistent stores? (§5.9)
- Does Ldc offer benefits in a production store? (§5.10)
- Does Ldc offer benefits for real-world traces? (§5.11)
- How does CBA benefit Ldc? (§5.12)

Setup. We use a cluster with three (and sometimes five) server replicas and many clients. Each machine has an Intel 10-Core E5-2640v4 CPU, 64GB DRAM, a 25Gb Mellanox ConnectX-4 NIC, and a 480GB SATA SSD. For all experiments, we use a 10M key-value dataset with 24B keys and 100B values (typical in applications [21]); cache percentage



(a) Uniform workload



(b) Other workload distributions

Figure 7. Read-only Workloads in Twig-kv: OrigCache vs. Ldc Under Different Workload Distributions and Cache Sizes.

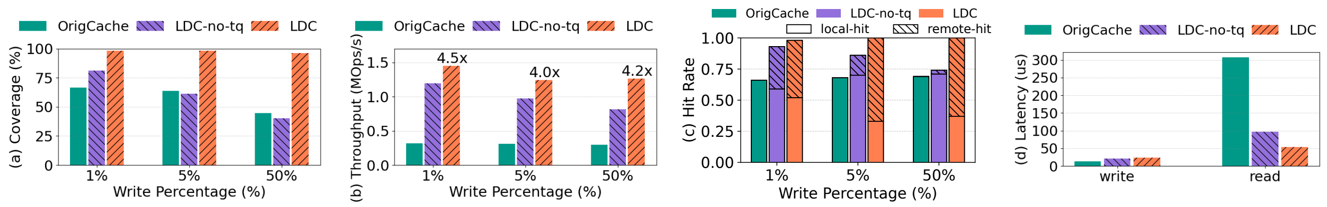


Figure 8. Read-Write Workloads in Twig-kv: Ldc vs. OrigCache and LDC-no-tq Under Different Write Percentages.

denotes the cache size on a single replica relative to the dataset size. In all systems, each client has a sticky session with one replica for reads and submits writes to the leader. This sticky-session approach mirrors real deployments. We evaluate Twig-kv, Craq-kv, and RethinkDB by comparing the Ldc version and the base version that uses original cache (referred to as OrigCache). We explore most microbenchmarks with Twig-kv. For running the real-world traces, we also built a simulator.

5.1 Read-Only Microbenchmark

We compare the Ldc and OrigCache versions of Twig-kv under read-only workloads. We explore different cache sizes and workload distributions.

Uniform. Figure 7(a) shows the result for the uniform workload. As shown in 7(a)(i), OrigCache has high CSF values. Ldc, in contrast, has zero CSF because when an object is present in a remote cache, it is not obliviously brought in at another cache. Ldc's CBA decides not to redundantly cache objects since coverage is important for the uniform workload. This enables Ldc to achieve higher throughput. Ldc's gains increase with higher cache sizes because it effectively covers more of the dataset in the cache, avoiding many disk accesses. In the 33.3% case, Ldc caches the entire dataset, improving throughput by 5.3× over the original system.

Zipfian. Figure 7(b) shows the result for a zipfian workload with 10% and 33.3% cache sizes. Here, although a few objects are very popular, the distribution has a long tail and thus increased coverage is still important. Ldc achieves this with low CSF-s. Ldc's CSF is not zero because the CBA decides to redundantly cache the few popular objects. Ldc offers notably higher throughput than OrigCache.

Hotspot. Figure 7(b) also shows the result for a hotspot workload where 80% reads are to 20% objects. With smaller cache size (10%), coverage is more important to contain the 20% hot objects across the caches. Ldc makes this correct choice (CSF=0), achieving better performance than OrigCache. With 33.3% caches, Ldc has more space after fitting the hot objects. Thus, Ldc chooses to redundantly cache many hot objects while still providing high coverage. This enables Ldc to achieve higher performance than OrigCache.

5.2 Mixed Read-Write Microbenchmark

We next examine how Ldc handles writes under different write percentages (1%, 5%, 50%). The workloads follow a zipfian distribution and each replica has 33.3% cache space. In addition to OrigCache, we also compare against LDC-no-tq, a Ldc variant that does *not* use the tiny queue and thus pollutes the main cache upon writes. For Ldc, we carve out 0.01% of the cache as the tiny queue. Thus, in Ldc, the main cache is 0.01% smaller than the baselines.

Figure 8 shows the results. OrigCache pollutes the main cache upon both reads and writes, leading to lower coverage (8(a)). LDC-no-tq does not pollute the cache on reads but it does so on writes. With a few writes (1%), LDC-no-tq improves coverage over OrigCache; this results in LDC-no-tq performing better than OrigCache, as shown in Figure 8(b). With more writes, LDC-no-tq's coverage becomes almost the same as OrigCache. Thus, LDC-no-tq's absolute performance decreases with increasing writes. However, LDC-no-tq continues to offer better performance than OrigCache because it can make the use of the remote caches, while OrigCache incurs a disk access upon a local miss. This is shown in Figure 8(c), where LDC-no-tq improves the overall hit rate over OrigCache by augmenting the local hits with remote hits.

Ldc improves over LDC-no-tq by also minimizing write-induced redundancy via quick demotions. Thus, Ldc has better coverage, which leads to higher overall hit rates and thus higher performance (e.g., 4.2× and 1.5× than OrigCache and LDC-no-tq, respectively with 50% writes).

We next show that Ldc's tiny queue has minimal impact on write performance. Figure 8(d) plots the read and write latency with 50% writes. The write latencies are generally lower than that of reads because writes are asynchronously replicated in Twig-kv and are absorbed in memory; reads can see disk or RDMA latencies. Ldc increases write latency over OrigCache because it writes via the tiny queue and thus writes may wait for evictions. However, even in this write-heavy workload, Ldc adds only 10μs over OrigCache. This is because although Ldc quickly demotes writes, it does so only selectively. Thus, when Ldc's CBA decides to cache extremely popular objects redundantly, repeated writes to those objects are absorbed in the main cache and need not wait for tiny-queue evictions.

Next, OrigCache and LDC-no-tq have much higher read latencies than Ldc. This is because writes in OrigCache and LDC-no-tq displace many objects; thus, more reads go to the disk. In contrast, Ldc does not displace objects in the main cache, achieving 5.8× and 1.8× lower read latencies than OrigCache and LDC-no-tq, respectively.

We also compare Ldc against a *write-around* variant, which is equivalent to using a zero-size tiny queue: if a written key is already cached, the write is absorbed in the main cache as usual; otherwise, the write bypasses the cache and goes directly to disk. Since a cache can only hold a limited number of keys, many writes whose keys are not cached, has to issue a write to the disk in the synchronous path, incurring high latency. In contrast, Ldc's tiny queue acts as an in-memory buffer that absorbs these writes and batches writes to disk asynchronously, avoiding synchronous disk access on the write path. Under our 50%-write workload, write-around achieves 3.2× lower throughput than Ldc and is even 1.3× slower than the OrigCache baseline that uses writeback method (graphs not shown). This shows the importance of the tiny queue.

5.3 Additional Microbenchmarks

We next run additional microbenchmarks with Twig-kv, where, we analyze (i) performance with replica scaling, (ii) cache sizes of OrigCache to match Ldc's performance.

5.3.1 Replica Scaling. We evaluate how OrigCache and Ldc perform with more replicas. Figure 9 shows the throughput with 3 and 5 replicas under a uniform read-only workload with each replica configured to have a 20% cache. When going from 3 to 5 replicas, both systems get 40% more memory and thus ideally should be able to collectively cache the entire dataset. However, OrigCache suffers from high cache redundancy, caching only 67% of the dataset with 5 replicas. This results in a modest 1.3× throughput increase in OrigCache when scaling from 3 to 5 replicas. In contrast, Ldc fully utilizes the cache space, realizing a 3.3× improvement for the same replica scaling.

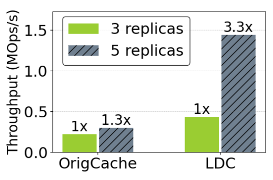


Figure 9. Replica Scaling.

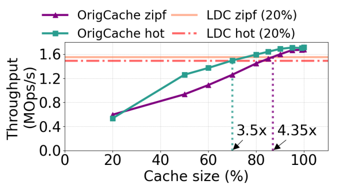


Figure 10. Matching Cache Size.

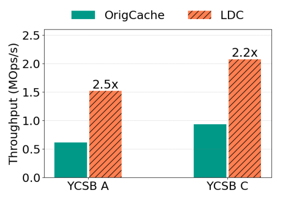


Figure 11. NVMe SSD Throughput.

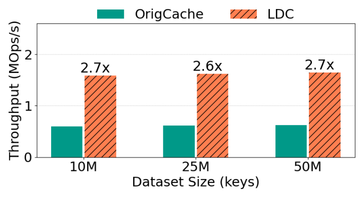


Figure 12. Dataset Scalability.

5.3.2 Cache Size to Match Performance. We analyze how much additional cache capacity OrigCache would require to match Ldc's performance. Figure 10 shows OrigCache's throughput with increasing percentage of cache sizes under zipfian and hotspot workloads. To meet Ldc's performance at 20% cache size with 5 replicas (horizontal lines in Figure 10), OrigCache requires significantly larger per-replica cache sizes: 4.35× for zipfian and 3.5× for hotspot. This shows the potential cost benefits in terms of required memory (for a target performance) that Ldc can bring.

5.4 Performance with fast NVMe Storage

While most experiments use SATA SSDs, we now evaluate if LDC can offer benefits even with faster nvme SSDs. Since LDC benefits come from avoiding disk accesses via remote cache hits over RDMA, one might wonder if LDC's benefits diminish with faster storage.

To evaluate this, we run Twig-kv with NVMe SSDs under YCSB C (100% reads) and YCSB A (50% writes) with 33.3% cache per replica and compare Ldc against OrigCache . As seen in Figure 11, for the read-only workload (YCSB C), Ldc provides a 2.2× throughput improvement, and for the write-heavy workload (YCSB A), Ldc achieves a 2.5× improvement. These results show that Ldc delivers significant performance gains even with fast NVMe storage.

5.5 Scalability with Larger Datasets

So far, all experiments use a 10M key-value pair dataset. To investigate whether LDC scales to larger datasets, we increase the dataset size to 25M and 50M key-value pairs while proportionately increasing the cache size so that each replica can still hold 33.3% of the dataset. As shown in Figure 12 LDC's performance gains remain consistent across larger dataset sizes.

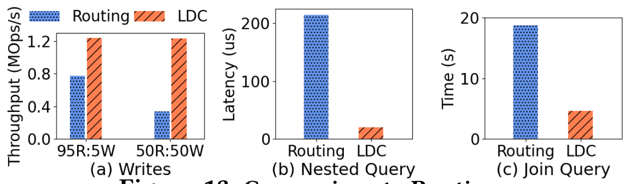


Figure 13. Comparison to Routing.

5.6 Comparison to Routing

We now compare Ldc to key-based routing, which routes reads to replicas based on a key's hash. Figure 13(a) shows the performance under mixed workloads. First, routing suffers from write-induced redundancy, displacing many object from the cache, resulting in many disk accesses for reads. Ldc, by avoiding write pollution, can serve most reads from the cache, offering better (1.6×–3.7×) throughput than routing.

We next run a read-only workload with nested queries of the form read(read(key)), where the final key is not known apriori. Thus, with routing, the client sends the query to a random replica, which executes it to find the final key. If the replica has the key in its cache, it returns the value; else, it accesses the key from the disk admitting the key into the cache if it is responsible for caching the key. Because these queries cannot be accurately routed, they incur many disk accesses. In contrast, in Ldc, the receiving replica executes the query and reads the inner and final keys from either local or remote caches without disk accesses. Thus, as shown in Figure 13(b), Ldc's latency is ~10× lower than routing.

We also evaluate a join query that combines two 1M-key tables: an *orders* table holding ⟨*order_id*, *customer_id*⟩ pairs and a *customers* table holding ⟨*customer_id*, *customer_name*⟩ pairs to obtain a result table of ⟨*order_id*, *customer_name*⟩ pairs. Unlike the nested-query example, here, we offload the query processing to the client (which is an option to do routing with opaque queries as discussed in §2.3). With routing, the client must issue two lookups, first to obtain *customer_id* from the order and then to fetch the customer name, incurring an extra round trip even when both accesses hit in the cache. Ldc executes the join at the receiving replica, fetching both keys from local or remote caches in a single request. Thus, as shown in Figure 13(c), Ldc outperforms routing by ~4.1×. Overall, Ldc offers significant benefits over routing under read-write and opaque read-only workloads.

5.7 Comparison to Other Related Approaches

We next compare Ldc to two other approaches that can possibly help reduce cache redundancy. The first is cooperative caching (CC), where a group of entities cooperatively cache data [29]. CC was originally designed for distributed-file-system clients, where clients access files from other clients' memory (instead of going to the server). We adapted CC to the *server* replicas to cooperatively cache the dataset. The second one is better caching policies at each replica that use the caches more effectively. One such state-of-the-art policy is that of s3-fifo [80, 81] that (like Ldc) uses quick demotions but to prevent admitting one-hit wonders into the local caches.

5.7.1 Cooperative Caching.

We implement CC, specifically the n-chance policy [29], in Twig-kv. With CC, the caches cooperate and can access each other. We give CC the RDMA advantage for remote accesses. N-chance makes two policy decisions: (i) upon a remote access, an object is always admitted locally, and (ii) a cache evicts singletons (i.e., unique items) to other caches, and such forwarding can happen N times. CC, in its original form, writes directly to the backend and simply invalidates the caches. Thus, in the CC version of Twig-kv, each replica directly writes to the disk and invalidates its cache; we call this CC-WI. We also build another version of CC, where writes are processed similar to the OrigCache, while using CC for reads; we call this variant CC-WB.

As shown in Figure 14(a), under the read-only workload, both CC variants perform worse than Ldc. First, the always-admit strategy in n-chance increases redundancy. While this policy is apt for clients that must optimize for their own performance, it is less suitable for servers which must optimize for the overall system performance. Second, the singleton evictions displace objects from the remote caches; while this strategy was acceptable for evicting objects to idle clients, it is unsuitable for servers because server caches are rarely idle. Under the mixed read-write workload, CC-WI performs poorly because writes incur disk access and it also invalidates the caches which causes read misses. Although CC-WB improves over CC-WI, it performs worse than Ldc because writes pollute the cache, decreasing coverage; this results in more disk accesses for reads.

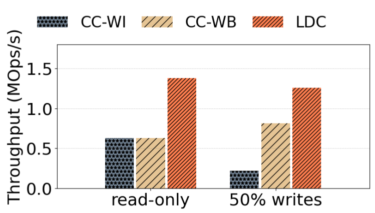
Overall, although CC cooperates across caches, it intends to balance a client's own performance and the overall good of the system [79]. In contrast, in the replicated storage setting, replicas can *always* optimize the overall good of the system instead of a single replica's performance. This difference results in CC's policies not performing well in our setting. Even if a CC policy favors the overall good of the system, it does not fundamentally handle write-induced redundancy, which is critical in our setting.

5.7.2 Better Local Cache Policies.

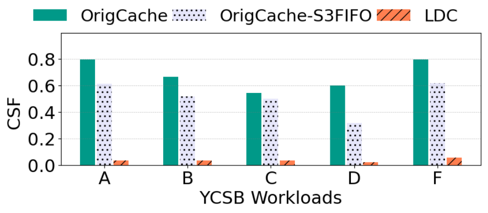
Using a simulator (§5.11), we simulate Twig-kv with its default policy (LRU) and s3-fifo, and compare it to Ldc. As shown in Figure 14(b), compared to LRU, s3-fifo does reduce redundancy. However, even with s3-fifo, the individual caches still operate in silos and thus cannot effectively prevent replicas from caching the same objects unlike Ldc.

5.8 Twig-kv: YCSB Macrobenchmark

We now evaluate Twig-kv under YCSB. As shown in Figure 15, first, in the read-only workload (C), Ldc offers 2.6× higher throughput than OrigCache. LDC-no-tq is also able



(a) Cooperative Caching vs. Ldc



(b) Better Local Policy (S3-FIFO) vs. Ldc

Figure 14. Comparison to Other Related Approaches.

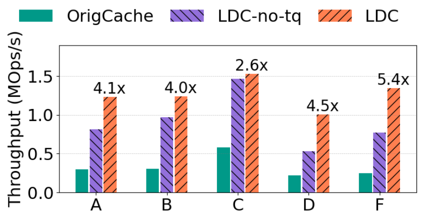


Figure 15. Twig-kv YCSB.

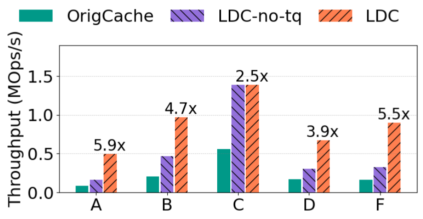


Figure 16. Craq-kv YCSB.

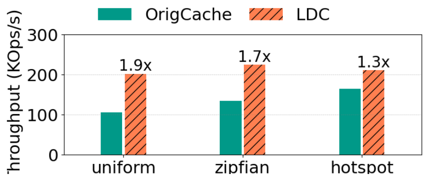
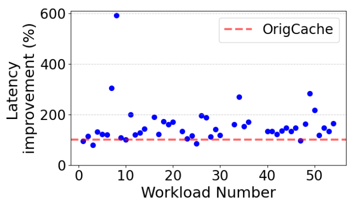


Figure 17. Benefits in RethinkDB.

Dataset	# traces	Avg. lat improve	Max lat improve	disk access reduction
Twitter [69]	47	1.57x	5.76x	50.07%
Meta [53]	2	1.46x	1.47x	40.04%
Alibaba [13]	1	1.39x	1.39x	32.6%

(a) Summary of Improvement under Traces



(b) Twitter Traces

Figure 18. Real-World Traces.

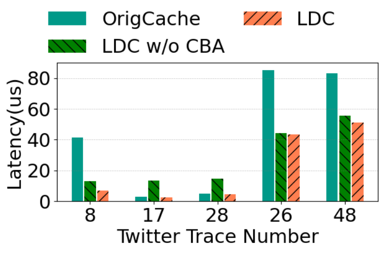


Figure 19. CBA Efficacy.

to match Ldc given that there are no writes. With a few writes (5% with B), although LDC-no-tq is better than OrigCache, it cannot match Ldc as writes introduce redundancy. This gap increases with more writes (A,F). In these write-intensive workloads, Ldc offers 5.4× better throughput than OrigCache. Although workload-D has only 5% writes, these writes are inserts, which increases the dataset size and thus more accesses go to disk, resulting in lower performance in general. Although Ldc incurs disk accesses, it still outperforms OrigCache and LDC-no-tq.

5.9 Benefits in a Strongly Consistent Store

We now show that Ldc can benefit systems that offer strong consistency. We compare the OrigCache and Ldc versions of Craq-kv under YCSB workloads. As shown in Figure 16, Ldc provides significant benefits even with a strongly consistent system, outperforming OrigCache by up to 5.9×. We also observe that Ldc offers such high throughput with low latencies. For example, Ldc offers 6.5× lower latency than OrigCache under YCSB-A. Note that the Ldc version of Craq-kv performs the version checks for ensuring strong consistency like OrigCache (§4).

5.10 Benefits in a Production Store

We next show that Ldc can benefit a production store by comparing the OrigCache and Ldc versions of RethinkDB under read workloads. As shown in Figure 17, Ldc improves throughput by up to 1.9×. While substantial, the benefits can be further improved as our RethinkDB implementation uses unoptimized synchronous RDMA calls as opposed to asynchronous calls in other implementations.

Memory overhead. We also measure the overhead required to realize Ldc in the production system. The additional memory required for maintaining the indexes for remote caches (8 bytes per 4K page for every remote replica) and CBA-related statistics adds less than 1% overhead at a replica.

5.11 Real-World Traces

We analyze whether Ldc offers benefits under 50 real-world traces shown in Figure 18(a). The traces have varying percentage of writes ranging from read-only to write-heavy (e.g., 80%-writes with trace-12 and 62%-writes with trace-13). The traces also vary significantly in terms of distribution (from highly uniform to extremely skewed).

To run this large trace collection, we built a simulator that simulates the OrigCache and Ldc versions of Twig-kv and measures local and remote hits, and disk accesses. Average latency is then calculated based on real-system latencies for these metrics. As shown in Figure 18(a), Ldc reduces average latency and disk accesses across traces significantly.

We examine the Twitter traces in more depth. We plot the latency improvement of Ldc over OrigCache in Figure 18(b). As shown, Ldc offers notable latency improvements across all the traces.

5.12 CBA Efficacy

We now analyze the efficacy of CBA. To do so, we compare Ldc against a version that does not implement CBA (Ldc-w/o-CBA). This variant does not admit remotely accessed objects locally. We compare Ldc against the variant without CBA and OrigCache using the above Twitter traces. Figure 19 shows the result for a subset of traces.

Trace-8 has 50% writes and thus OrigCache performs poorly due to cache redundancy from writes. In this trace, some objects are very popular. The Ldc-w/o-CBA variant handles writes but it does not redundantly cache the popular objects. Ldc improves over Ldc-w/o-CBA as it caches the popular objects redundantly.

Trace-17 and Trace-28 are highly skewed read-heavy workloads. OrigCache's high redundancy helps here and thus it performs well. Ldc-w/o-CBA does not cache popular objects redundantly and thus performs poorly compared to even OrigCache. In contrast, Ldc's CBA correctly decides to redundantly cache the popular objects, matching the performance of OrigCache.

Finally, Trace-26 has 29% writes and Trace-48 has 35% writes, but they are not skewed. Thus, Ldc-w/o-CBA performs well by increasing cache coverage, while OrigCache does not due to high cache redundancy. Ldc matches the performance of Ldc-w/o-CBA in this case since CBA correctly decides to not cache the remote objects. Overall, Ldc's CBA helps balance cache coverage and redundancy, offering significant benefit.

6 Related Work

We describe prior work and place our work in context.

External Caches. As we discussed, there has been a rich body of work on optimizing external caches [7, 52, 57, 59]. Our focus is new: embedded caches in the replicated-storage servers; applying routing from external caches to our setting leads to problems (§5.6, §2.3). External caches use selective replication [43, 45] to address load imbalance with hot keys. Ldc's CBA has similar goals of redundantly caching popular objects but to avoid remote accesses.

Cooperative Caching (CC). A body of work has focused on cooperatively caching data among a group of entities, which has been explored in distributed file systems [29, 65], web caches and CDNs [36, 79], and chip multiprocessors [18, 24, 25, 42]. Compared to this body of work, our context and aims are different: to effectively manage the embedded caches in replicas of a KV store or database. This context presents new challenges; for example, CC approaches do not consider writes carefully, while it is critical to do so in our setting. Further, adapting client-centric CC algorithms to cooperate among servers, even under read workloads, results in poor performance as the policies of CC are less suitable for cooperation among servers (as we showed in §5.7).

Exclusive Caching. Prior work in hierarchical caches strive to ensure exclusivity across different layers of the stack (e.g., file-system and disk caches) [17, 26, 38, 50, 74, 77, 78, 83]. While this goal is similar to Ldc's goal, prior work mostly focuses on single-node storage. These techniques are complementary to Ldc: they can be applied within each replica, while Ldc can minimize cache redundancy across replicas.

RDMA-Based Systems. A vast body of work has leveraged RDMA in key-value stores [34, 49, 54, 71, 72, 82, 84], memory disaggregation [9–12, 14, 39, 44, 47, 64, 85], and cluster caches [66, 67]. A key focus in some of these systems is how to efficiently use RDMA (e.g., reducing roundtrips [71]). These RDMA techniques are complementary to our core contributions and can be used to improve Ldc's implementation further. For example, one could use RACE hashing [85] to index remote caches more efficiently. More broadly, while RDMA makes remote accesses in Ldc efficient, the Ldc idea is general and even an RPC-based design is possible.

7 Conclusion

This paper studies if replicated storage systems effectively utilize the embedded caches. Our study reveals that embedded caches across replicas exhibit high redundancy, wasting precious in-memory cache space. To address this problem, we introduce logically disaggregated cache (Ldc), a new architecture to better manage embedded caches. Ldc unifies the individual caches to form a single logical cache, which improves cache efficiency and thus performance. Ldc offers a new way to derive maximum benefit from in-memory caches, an expensive resource in datacenters.

Acknowledgments

We thank our shepherd Ashvin Goel and the anonymous EuroSys '26 reviewers for their insightful comments. We also thank the other DASSL members for their discussions. This work was funded by NSF CNS-2340218 and CNS-2339784, an IIDAI grant, a gift from TigerBeetle. We also thank Cloudlab for providing a great environment to run our experiments.

References

[1] HBase Read Replicas. https://docs.cloudera.com/runtime/7.3.1/hbase-high-availability/topics/hbase-read-replicas.html, 2021.

[2] HBase Region Servers. https://nag-9-s.gitbook.io/hbase/hbase-architecture/region-servers, 2021.

[3] About read replicas. https://cloud.google.com/memorystore/docs/redis/about-read-replicas, 2025.

[4] MongoDB Read Concern "local". https://www.mongodb.com/docs/manual/reference/read-concern-local, 2025.

[5] SQLite B-Tree Module. https://sqlite.org/btreemodule.html, 2025.

[6] What Is SQLite? https://www.sqlite.org/index.html, 2025.

[7] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer:Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[8] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory Disaggregation: Why Now and What are the Challenges. *SIGOPS Oper. Syst. Rev.*, 57(1):38–46, jun 2023.

[9] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory disaggregation: why now and what are the challenges. 57(1):38–46, jun 2023.

[10] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Regions: A Simple Abstraction for Remote Memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, July 2018.

[11] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857, 2020.

[12] Hasan Al Maruf and Mosharaf Chowdhury. Memory Disaggregation: Advances and Open Challenges. 57(1):29–37, jun 2023.

[13] Alibaba. Alibaba Block Traces. https://github.com/alibaba/block-traces, 2022.

[14] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of the EuroSys Conference (EuroSys '20)*, Crete, Greece, April 2020.

[15] Apache. Cassandra. http://cassandra.apache.org/, 2025.

[16] Apache HBase. Apache HBase. https://hbase.apache.org/, 2019.

[17] Lakshmi N Bairavasundaram, Muthian Sivathanu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. *ACM SIGARCH Computer Architecture News*, 32(2):176, 2004.

[18] Bradford M Beckmann, Michael R Marty, and David A Wood. ASR: Adaptive selective replication for CMP caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 443–454. IEEE, 2006.

[19] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[20] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-backup Approach. *Distributed systems*, 2, 1993.

[21] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020.

[22] Cassandra. How is the consistency level configured? https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlConfigConsistency.html, 2022.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. pages 15–15, 2006.

[24] Jichuan Chang and Gurindar S Sohi. "cooperative caching for chip multiprocessors". *ACM SIGARCH Computer Architecture News*, 34(2):264–276, 2006.

[25] Jichuan Chang and Gurindar S Sohi. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 402–412, 2007.

[26] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX, June 2003. USENIX Association.

[27] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvmekey-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Online, July 2020.

[28] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IN, June 2010.

[29] Michael D Dahlin, Randolph Y Wang, Thomas E Anderson, and David A Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, CA, November 1994.

[30] DataStax. How caching works. https://docs.datastax.com/en/cassandra-oss/2.1/cassandra/operations/ops_how_cache_works_c.html, 2015.

[31] DataStax. Load Balancing and Failure Handling Policies. https://docs.datastax.com/en/developer/python-driver/3.8/api/cassandra/policies/index.html, 2018.

[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 41(6), October 2007.

[33] Ola H Diserud and Frode Ødegaard. A Multiple-Site Similarity Measure. *Biology letters*, 3(1):20–22, 2007.

[34] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[35] eBay. NuRaft. https://github.com/eBay/NuRaft, 2019.

[36] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, jun 2000.

[37] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Strong and Efficient Consistency with Consistency-aware Durability. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020.

[38] Binny S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better Than Demotions. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[39] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th Symposium on Networked Systems Design*

and Implementation (NSDI '17), Boston, MA, March 2017.

[40] Chuanxiong Guo. Rdma in data centers: Looking back and looking forward. *Keynote at APNet*, 2017.

[41] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, New York, NY, United States, August 2016.

[42] Enric Herrero, José González, and Ramon Canal. Distributed cooperative caching. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 134–143, 2008.

[43] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[44] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. Aceso: Achieving Efficient Fault Tolerance in Memory-Disaggregated Key-Value Stores. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '24)*, Austin, Texas, October 2024.

[45] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014.

[46] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.

[47] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and Highly Available Remote Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[48] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[49] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *Proceedings of the 21th USENIX Conference on File and Storage Technologies (FAST '23)*, Santa Clara, CA, February 2023.

[50] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, page 9, USA, 2005. USENIX Association.

[51] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[52] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, February 2019.

[53] Meta CacheLib. Evaluating SSD hardware for Facebook workloads. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval/, 2021.

[54] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 103–114, USA, 2013. USENIX Association.

[55] MongoDB. MongoDB. https://www.mongodb.org/, 2025.

[56] MongoDB. MongoDB WiredTiger. https://docs.mongodb.org/manual/core/wiredtiger/, 2025.

[57] 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. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, April 2013.

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

[59] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[60] RethinkDB. RethinkDB. https://www.rethinkdb.com/, 2023.

[61] RethinkDB. Scaling, sharding and replication. https://rethinkdb.com/docs/sharding-and-replication/, 2025.

[62] RethinkDB. Understanding RethinkDB memory requirements. https://rethinkdb.com/docs/memory-usage, 2025.

[63] RQLite. Rqlite. https://rqlite.io/, 2025.

[64] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[65] Prasenjit Sarkar and John Hartman. Efficient Cooperative Caching Using Hints. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*, Seattle, WA, October 1996. USENIX Association.

[66] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 675–691, New York, NY, USA, 2023. Association for Computing Machinery.

[67] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM Conference*, Virtual, August 2021.

[68] Jeff Terrace and Michael J Freedman. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, CA, June 2009.

[69] Twitter. Twitter Cache Trace. https://github.com/twitter/cache-trace, 2020.

[70] VMware. SplinterDB. https://splinterdb.org/, 2022.

[71] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[72] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, November 2018.

[73] WiredTiger. https://source.wiredtiger.com/develop/arch-cache.html, 2024.

[74] Theodore Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.

[75] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Event, February 2021.

[76] Yi Xu, Henry Zhu, Prashant Pandey, Alex Conway, Rob Johnson, Aishwarya Ganesan, and Ramnatthan Alagappan. IONIA: High-Performance Replication for Modern Disk-based KV Stores. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, Santa Clara, CA, February 2024. USENIX Association.

[77] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Transactions on Computer Systems (TOCS)*, 29(2):1–51, 2011.

[78] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-All replacement for a multilevel cache. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. USENIX Association.

[79] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative Caching with Return on Investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2013.

[80] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 70–79, 2023.

[81] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo Queues Are All You Need for Cache Eviction. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*, Koblenz, Germany, October 2023.

[82] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. Redy: Remote Dynamic Memory Cache. *Proc. VLDB Endow.*, 15(4), December 2021.

[83] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. 15(6):505–519, jun 2004.

[84] Tobias Ziegler, Carsten Binnig, and Viktor Leis. ScaleStore: A Fast and Cost-Efficient Storage Engine Using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22)*, Philadelphia, PA, June 2022.

[85] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided {RDMA-Conscious} extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29, 2021.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code, configuration files, and scripts to build, configure, and run Twig-kv — our eventually-consistent replicated KV store — with and without LDC integration. The artifact allows evaluators to:

- Build the Twig-kv system and all its dependencies on a CloudLab cluster.
- Run the system under two modes: **OrigCache** (baseline — isolated per-replica caches with cache-unaware routing) and **LDC** (our proposed system — logically disaggregated cache with one-sided RDMA).
- Evaluate across multiple key distributions (uniform, Zipfian, hotspot) generated via the included YCSB workload generator.
- Observe throughput, latency, and cache hit-rate differences between the baseline and our proposed system.

The artifact includes automated init/setup scripts, experiment orchestration, and workload generation.

A.2 Description & Requirements

A.2.1 How to access. The artifact is publicly available at: `https://github.com/dassl-uiuc/Logically-Disaggregated-Caches`

A.2.2 Hardware dependencies. A cluster of 7 machines (3 clients + 3 servers + 1 optional control node; a server node can double as the control node), each with an RDMA-capable NIC (e.g., Mellanox ConnectX-4/5), connected via InfiniBand or RoCE. We recommend using CloudLab `XL170` machines or similar.

Authors' environment: CloudLab `XL170` machines with Intel 10-Core E5-2640v4 CPU, 64 GB DRAM, 25 Gb Mellanox ConnectX-4 NIC, 480 GB SATA SSD per machine.

A.2.3 Software dependencies. All dependencies are installed automatically by the `init.sh` script:

- Ubuntu 22.04 (tested), Linux 5.4+
- apt: build-essential cmake gcc meson ninja-build libboost-all-dev libgoogle-glog-dev libgflags-dev libnuma-dev libtbb-dev liburing-dev nlohmann-json3-dev (full list in `setup.py`)
- pip: numpy matplotlib beautifulsoup4 asyncssh lxml tqdm
- Third-party submodules: MachNet (networking), RDMA (Infinity verbs), YCSB (workload generator), among others.

A.2.4 Benchmarks.

- **YCSB**: Default 24B key, 100B value KV pairs. Workloads generated via `create_ycsb_workload.sh` with three distributions: Uniform, Zipfian, and Hotspot.

A.3 Set-up

1. We have provisioned a CloudLab cluster for the artifact evaluation. (To use your own cluster, follow the instructions in the `README.md`.)
2. Run `./init.sh` to install Python and RDMA dependencies.
3. Run `./init_compile.sh` to compile the system on all nodes (first time only).
4. Generate the dataset: `./create_ycsb_workload.sh` produces trace files for the desired distribution. Refer to the `README.md` for more details.

A.4 Evaluation workflow

A.4.1 Major Claims.

- **(C1)**: LDC (our proposed system) improves throughput and reduces latency compared to the baseline (isolated per-replica caches with cache-unaware routing) in Twig-kv under Uniform, Zipfian, and Hotspot workloads.

A.4.2 Experiments. Each experiment uses `run.sh`; follow the instructions in the `README.md` to run the desired experiment.