Flex-KV: Enabling High-performance and Flexible KV Systems

Amar Phanishayee[1], David G. Andersen[1],
Himabindu Pucha[2], Anna Povzner[3], Wendy Belluomini[3]

[1]*Carnegie Mellon University,* [2] *Violin Memory,* [3]*IBM Research Almaden*

ABSTRACT

Even within one popular sub-category of ”NoSQL” solutions – key-value (KV) storage systems – no one existing system meets the needs of all applications. We question this poor state of affairs. In this paper, we make the case for a flexible key-value storage system (Flex-KV) that can support both DRAM and disk-based storage, can act as an unreliable cache or a durable store, and operate consistently or inconsistently. The value of such a system goes beyond ease-of-use: While exploring these dimensions of durability, consistency, and availability, we find new choices for system designs, such as a cache-consistent memcached, that offer some applications a better balance of performance and cost than was previously available.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design–*Distributed Systems*; D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability–*Fault-tolerance*;

General Terms: Design, Performance, Reliability

Keywords: Design, Performance, Cluster Computing

1 Motivation

We are witnessing an explosion of “NoSQL” storage systems, used by companies ranging from startups to industry giants including Amazon, Google, Facebook, and Twitter [4, 9, 11]. Their popularity has resulted in cloud service providers offering NoSQL key-value (KV) systems as building blocks for applications. Each system provides slightly different semantics or is optimized for subtly different use cases. This situation is tragic: It impedes the flexibility of cloud providers and developers by forcing them to commit to a particular model, which they can change only by switching to an entirely different system. It furthermore misses numerous opportunities for worthwhile designs that fall “in-between” existing storage system design choices. In this paper, we argue that it is possible to create *one* storage system that can meet the needs of all of these applications.

Some applications or operations demand synchronous, durable replication; others favor availability over consistency; and for yet others, the cost of such safety is orders of magnitude too expensive, making it impossible to meet latency or throughput requirements. These requirements sit on a multi-dimensional continuum, with the breadth of NoSQL KV systems testifying to the value of finding a design and implementation well matched to one's requirements. Flash memory and purely in-memory datastores add yet more tradeoffs between sequential and random read/write performance, durability, and power consumption, which further complicates the design space for data storage systems.

Unfortunately, this demand places system designers in a bind: Do they run multiple stores, each operating at maximum efficiency, or do they optimize instead for system complexity by avoiding the need for multiple codebases, vendors, configurations, and so on? We argue that placing systems designers in this bind is unreasonable and, our work suggests, unnecessary. Instead, we show that a KV architecture designed right can easily be configured to support many points along this continuum, from weakly-consistent, non-replicated caches [9] to strongly-consistent, durable disk-backed key-value stores [6].

This paper argues that a design based on simple chain-based replication enables such a flexible architecture. We introduce Flex-KV, a configurable key-value storage system that uses chain-based replication to effectively support a wide range of application requirements. Flex-KV can support DRAM, disk, or Flash-based storage, can support homogeneous or heterogeneous replica chains that can act as an unreliable cache or a durable store, and can trade strong data consistency for higher performance by varying the communication protocols between the replicas in the chain and selecting the query replica. The value of such a system goes beyond ease-of-use: While exploring these dimensions of durability, consistency, and availability, we find new choices for system designs supported by replica chains, such as a cache-consistent memcached, that offer some applications a better balance of performance and cost than was previously available. The schemes and preliminary results we discuss in this paper use a simple hash-style key-value system, but we believe that core design ideas apply to other storage systems as well.

2 KV Design Space

Systems designers today must pick a particular implementation to meet their application's needs. Current key-value systems differ in three major ways:

Durability What happens to data when the entire KV system is rebooted? Many key-value systems are used as a DRAM *cache* backed by relational databases or storage systems, e.g., the popular *memcached* system. On a cache miss, data is fetched from the backend and is then cached in the key-value system for future use. Updates (puts and deletes) are committed to the backend storage to guarantee data durability.

Other key-value systems act as the primary persistent store without a backend database, e.g., *MemcacheDB* or Redis [13]. There exist important differences in what data these systems may lose upon failures: Some sacrifice performance to write data synchronously to disk, and others favor a higher-performing asynchronous model.

Consistency Some applications may tolerate trading consistency semantics for performance and availability, e.g. Dynamo [4]. A strongly consistent system has the same value across replicas for all keys. Weakly consistent systems allow replicas to return older or different values for any key. For notational clarity, we permit a strongly consistent system that does not guarantee durability, in the face of failures, to either return "failure" or an older value for a key, *if* it correctly informs the client that the value is stale.

Availability in the presence of failures Failures affect data recoverability, system response time, and throughput. We classify availability as:

1. Data Availability (DA): What fraction of nodes can fail before data loss, with a given replication factor?
2. Performance Availability (PA): On a failure, how long does it take until performance is back to that when there were no failures?

2.1 Example: A Memory-efficient Configuration

Existing KV caches offer two extreme options: non-replicated configurations are memory efficient but suffer from poor performance availability; in-memory replicated schemes have higher memory overhead but better performance availability. To bridge this extereme divide, we propose a new design choice: A DRAM-based key-value store that provides high *performance availability* in the face of failures without the memory overhead of the simple replication strategies used today.

Sites such as LiveJournal, Facebook, and Twitter use *memcached* to support a read-mostly workload of millions of page views every day. Because of the huge performance gap between the cache (100,000s of queries per second) and the backing database (1,000s of queries per second), they devote terabytes of DRAM so that nearly all queries are served from cache. Writes invalidate entries in *memcached*, and directly update the database for persistence. Subsequent queries are then cached in *memcached* after being fetched from the database.

The large gap between cache and DB performance means that a cache node failure imposes a sudden high load on the backend database—higher than it can handle, degrading performance or even causing an entire site failure [12]. These sites require high *performance-availability*: they must continue to serve queries at in-memory speeds in the presence of failures.

Non-replicated and in-memory replicated systems offer two extreme options, with trade-offs between recovery time and memory overhead, shown as the "M", "M-M", and "M-M-M" configurations in Figure 1. In-memory replication, supported by systems such as memcached [8], repcached, and Gear6 [5], improves performance availability at the cost of at least twice as much DRAM, already measured in terabytes. The non-replicated system suffers long recovery time because it must read all data into cache from the backend database, potentially requiring random reads from disk.

Disk-backed cache with fast restore (M–D). Instead of naively replicating in-memory, an alternate design logs updates to disk on the replica ("M-D" configurations in Figure 1). If the primary fails, the system can rapidly stream the logged cache contents from disk to memory. Synchronous updates can be sped up by buffering updates at the replica before flushing them to disk asynchronously, giving rise to the variant M–$(M_b \ldots D)$ – a mechanism used in RAMCloud [10]. Alternatively, when used merely as a cache, it is acceptable to lose a small recent window of writes, and so updates can be propagated asynchronously to the disk replicas. To avoid inconsistency, however, it is necessary to synchronously invalidate entries on the replica (in-memory for speed).

This combination is memory efficient while still serving both reads and writes at memory speeds – a cache consistent memcache.

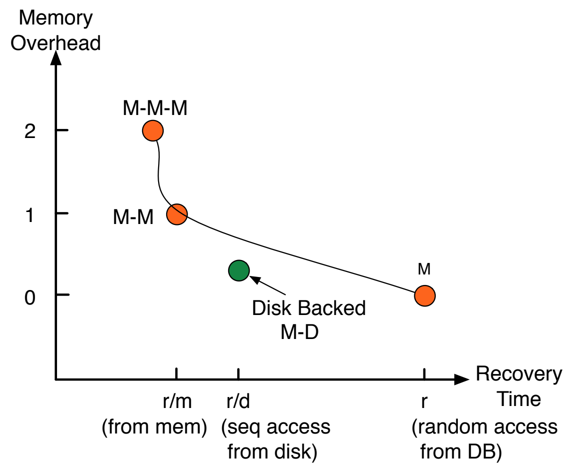

Figure 1: Disk backed replicas offer better tradeoffs between memory overhead and performance availability compared to options available today.

3 Flex-KV – A Flexible KV System

To implement all the configurations described in the previous section, a KV architecture must be able to: support heterogeneous replicas (e.g., disk, Flash, memory, etc.); direct queries and inserts appropriately (e.g., both queries and inserts to the in-memory replica for high performance, or queries to the tail and updates to the head for strong consistency); send invalidations and updates as configured; flexibly choose whether to send them synchronously or asynchronously; and optionally consult an invalidation table while applying updates on recovery.

Chain-based replication provides an effective mechanism to implement these options. Flex-KV uses replica chains on a consistent hashing ring. Consistent hashing with virtual nodes balances load across the backends and reduces repair time when nodes fail or new nodes join the system. Our prior work, FAWN-KV [1], uses a similar approach, but it supports only synchronous, durable, and consistent updates to Flash-based replicas, while routing queries to the tail of a replica chain. Flex-KV supports the range of application requirements, listed in the previous section, by supporting:

Replica types: Flex-KV supports different replica types that expose a common storage interface; examples include in-memory replicas (M), disk based logs (D) and buffered logs ($M_b \ldots D$). Flex-KV supports the addition of new datastores as long as they adhere to the storage interface. On each node, it is easy to combine different types of datastores by chaining their interfaces together, as is done in Anvil [7]. All update operations in Flex-KV are log-structured thereby ensuring high performance on different storage devices.

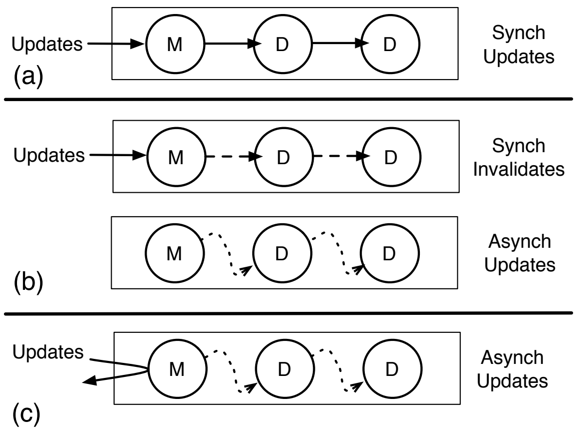

Figure 2: Three options for propagating updates through a chain of replicas.

Heterogeneous replica chains: Chaining of replicas [15] provides the basis for a variety of system options (e.g., creating M–D replica chains). Figure 2 shows several example configurations. Updates arrive at the head of the chain and propagate through the chain to the tail, as in chain replication. Unlike conventional chain replication where queries are served by the tail of the chain, Flex-KV allows queries to go to different nodes in each replica chain. For example a high-performance configuration may wish to direct reads and writes to a memory based replica. This flexibility "breaks" the simple structure of chain replication to also support a more conventional primary-backup replication style. Supporting synchronous insertions with read-from-head behavior requires more flexible configuration of when nodes will respond to queries they have already processed, marking un-acknowledged but received writes as tentative. Flex-KV hides the work of allocating replicas and managing the topology, and separates these functions from, e.g., the implementation of the replica's per-node storage.

Flexible update "plumbing" between replicas: The system separates update propagation and invalidation, and allows each to be delivered synchronously or asynchronously. We examine three ways to "plumb" replicas together. Key to these options are the ability to add asynchrony between purely memory-based replicas and disk replicas, to allow the system to operate with memory latency, not disk latency. Flex-KV can send updates using:

- **Synchronous Updates (SU)**: Figure 2(a) shows synchronous update propagation through a chain, creating three consistent replicas. Updates succeed only if all replicas are updated.

- **Asynchronous Updates with Synchronous Invalidations (AUSI)**: An update succeeds only if the primary commits the updated value and all secondary replicas receive invalidations (Figure 2(b)). Secondary replicas maintain an in-memory invalidation map. Updates are sent in batches from the primary to secondary replicas. Secondary replicas can clear their invalidation map after applying a batch of updates. This scheme enables coherent memory caches that recover from disk (e.g., the example in the previous section).
- **Asynchronous Updates (AU)**: An update succeeds if the primary replica commits the updated value. Secondary replicas receive either individual updates or a batch of updates asynchronously. (Figure 2(c))

Replication factor: Flex-KV allows configuring the system with an arbitrary replication factor. Flex-KV maintains this replication factor as long as it is possible to do so. On a node addition the replication factor of the chain it joins goes up by one and it is restored by relinquishing the current tail replica. On a node failure, the replication factor of the affected chain goes down by one and it is restored by recruiting a new tail for this chain. To ensure high performance, node additions and removals are non-blocking operations.

In summary, Flex-KV supports many different key-value system configurations using four simple knobs (Figure 3):

1. Replication Factor;
2. Replica Type: Memory, Disk, Flash, Buffered Log, etc.;
3. Update mechanism: SU, AUSI, and AU;
4. Query node: Replica to issue a read request to.

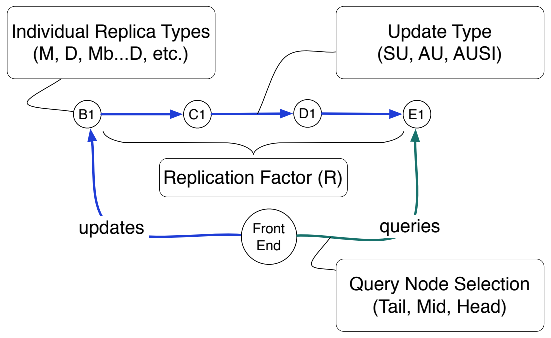
Individual Replica Types (M, D, Mb...D, etc.)
Update Type (SU, AU, AUSI)
B1
C1
D1
E1
Replication Factor (R)
updates
Front End
queries
Query Node Selection (Tail, Mid, Head)

Figure 3: Flex-KV supports many different key-value system configurations using four simple knobs.

Each configuration of our Flex-KV implementation trades between durability, memory overhead, performance, and recovery time. Figure 4 shows this tradeoff for five different Flex-KV configurations. In this experiment each backend node stores 15,000 KV pairs with 1KByte values. In-memory replication (M–M) uses twice as much memory as its unreplicated counterpart, but recovers instantly from a node failure. Heterogeneous replica chains (e.g., M–D) are memory efficient and recover more rapidly from node failures than an unreplicated node, and their recovery time is bound by sequential disk scan speeds. Schemes with synchronous updates (SU) are slow when they involve synchronous disk writes.

In the next two sections, we systematically vary the knobs exposed by Flex-KV (Tables 1, 2). Each cell in those tables represents a unique KV design, to illustrate the coverage of design options provided by Flex-KV's configurability—some choices are similar to currently available point solutions, and some offer new tradeoffs.

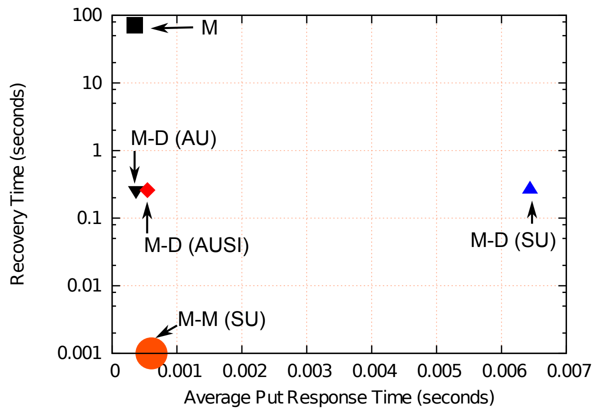

Figure 4: Memory overhead, put latency, and recovery time for different key-value configurations when using Flex-KV. The size of the points indicate memory overhead: M–M uses twice as much memory as its unreplicated counterpart.

3.1 Key-Value systems as *caches*

We start with KV systems with a backing database. When used with an external database, a key-value storage system (e.g., memcached) does not need to write synchronously to disk for persistence. It may, however, need replication for high *performance-availability*. Table 1 compares systems constructed with different replica types and update propagation mechanisms using four metrics: Read speed, update speed, memory overhead, and performance availability.

The horizontal axis in table 1 compares the results of using different plumbing between replicas. In general, synchronous updates provide consistency: A replica can fail and the data is still available in cache, but they bound the system performance to that of the slowest replica in the chain. Asynchronous updates lack consistency, but

Configuration	Synchronous Updates (SU) ***Consistent***	Async Updates, Synch Invalidations (AUSI) ***Consistent***	Asynchronous Updates (AU) ***Weakly Consistent***
In-memory Replication M–M +Backing Database(D) ***Memory inefficient***	Updates: Fast (slower than non-replicated systems) PA: Instant recovery Example: Gear6	Updates: Faster for large values PA: Nearly Instant (some cache misses)	Updates: Fast PA: Instant Example: repcached [14]
Disk backed cache M–D +Backing Database(D) ***Memory efficient***	Updates: Slow due to disk flush at replica (buffer for speed: M–($M_b \dots D$)) PA: Disk scan	Updates: Faster for large values (quick in-memory invalidations) PA: Disk scan (some cache misses)	Updates: Fast PA: Disk scan

Table 1: KV configurations with a backing database providing durability. We show configurations with one secondary replica, but the characteristics hold true for similar configurations with *n* secondary replicas.

allow the system to perform at the speed of the fastest replica. AUSI updates provide consistency without data-loss, and decouple performance, making them the best choice when acting as a cache.

Durability: A backing database in all configurations ensures data durability across the board.

Memory overhead: All configurations with in-memory replication have high overhead. M–D configurations with synchronous invalidations need only store invalidations in memory, and so their overhead is determined by the frequency of the asynchronous updates and the workload's update rate.

Update performance: Asynchronous updates are faster than synchronous updates, but this speed advantage also depends on the write cost at the next replica; even memory-to-memory replicas may be faster using AUSI updates if the updates are large. Disk-based replicas benefit more from asynchrony.

Performance Availability (PA): Configurations with both SU and AU recover almost instantaneously on failure. M–M recovers almost instantaneously. M–D is slower than M–M but is still much faster than random queries to the backing DB. The reason these configurations are not as rapid as M–M during recovery is because M–D involves a sequential scan of the log on the disk. The performance availability of configurations with AUSI are sightly worse than the corresponding configurations of SU or AU, because they involve applying invalidations and might incur cache misses for accesses of those key-value pairs that are invalidated. Figure 4, for example, shows the best case recovery time for the AUSI scheme where there are hardly any cache misses on recovery.

3.2 Key-Value systems as *stores*

Without a backing DB, most configurations retain the same properties, with one crucial difference: Durability. Configurations using only DRAM are not durable (Table 2), but neither does a configuration with a disk replica guarantee durability: In the table, only configurations with synchronous disk writes, either by starting with a disk, e.g., D–D, or using SU propagation to disk, e.g., M–D, are fully durable.

Configurations using AU and AUSI schemes with a disk replica have *window loss durability*: the system might have an older version of the value for some key, or no value at all, if there is a failure of a primary before updates are propagated to replicas. AUSI invalidations only provide *correct* durability if the invalidations are written synchronously to disk; this does not matter in the cache case, because if both the memory and disk replica fail, the system can recover from the database with some loss of performance. Configurations using AUSI can inform clients that the system lacks the latest update in case of such a failure. Using fully asynchronous updates risks undetected inconsistency.

Window loss durability may suffice for situations in which rare instance of stale data *could* be acceptable, e.g., for data such as web counters or "last visitors" lists, but where complete data loss over all time would not.

4 A Flexible Future

A first question our work raises is how users of storage systems should choose a configuration. Although orthogonal to the arguments of this paper, this question needs to be addressed in the future, not only for Flex-KV, but also for storage systems that provide different guarantees and tradeoffs at large.

A second important future question is how to extend

Configuration	Synchronous Updates (SU) ***Consistent***	Async Updates, Synch Invalidations (AUSI) ***Consistent***	Asynchronous Updates (AU) ***Weakly Consistent***
M–M ***Memory Inefficient***	Not Durable Example: Gear6, scalaris	Not Durable	Not Durable Example: repcached
M–$(M_b \ldots D)$ ***Memory Efficient***	Window-loss Durable Example: RAMCloud	Window-loss Durable (Cognizant of loss)	Window-Loss Durable
M–D ***Memory Efficient***	Durable	Window-loss Durable (Cognizant of loss)	Window-Loss Durable Example: Redis
Disk or Flash based *D–D*	Durable Example: FAWN-KV, Hibari	Window Loss Durable (Cognizant of loss)	Window Loss Durable Example: Tokyo Tyrant

Table 2: Comparison of different KV configurations without a backing database, all supported by Flex-KV.

the Flex-KV idea to encompass more "NoSQL" designs. Dynamo [4] shows one additional design axis: it trades consistency for partition tolerance. Systems such as Redis [13] (a data structure server supporting strings, hashes, lists, sets and sorted sets); BigTable [2] (column-oriented storage); and MongoDB [3] (document-based storage) all demonstrate the value of richer data models.

Creating "one store for all" is difficult, and it is likely that no one system can truly meet the needs of all users. However, our progress designing Flex-KV suggests that the right set of configuration and coupling primitives can make structured storage systems able to satisfy a wide variety of performance, consistency, and durability requirements. We believe that future research addressing these challenges is important.

References

[1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Seattle, WA, Nov. 2006.

[3] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, 2010.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[5] Gear6. Cache replication with Gear6 Web Cache. http://www.gear6.com/sites/gear6.com/files/Cache%20Replication%20Solution%20Brief%20Final.pdf.

[6] I. Gemini Mobile Technologies. Hibari: A Whitepaper. http://www.geminimobile.com/developers-center/white-papers/hibari-whitepaper-v1.0.pdf.

[7] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with Anvil. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[8] Memcache Failover FAQ. How does memcached handle failover? http://code.google.com/p/memcached/wiki/FAQ#How_does_memcached_handle_failover?

[9] Memcached. A distributed memory object caching system. http://memcached.org/, 2011.

[10] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[11] Project Voldemort. A distributed key-value storage system. http://project-voldemort.com.

[12] reddit. May 2010 "State of the Servers" report. http://blog.reddit.com/2010/05/reddits-may-2010-state-of-servers.html.

[13] Redis. A data structure server. http://redis.io/documentation.

[14] repcached. Add data replication to memcached. http://repcached.lab.klab.org/.

[15] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.