ASYNCHRONOUS WRITES IN CACHE AUGMENTED DATA STORES

by

Hieu Nguyen

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

May 2019

Abstract

This dissertation explores the processing of writes asynchronously in a cache augmented data store while maintaining the atomicity, consistency, isolation, and durability (ACID) properties of transactions. It enables the caching layer to dictate the overall system performance with both read-heavy and write-heavy workloads, motivating alternative implementations of a write-back policy. A write-back policy buffers writes in the cache and uses background threads to apply them to a data store asynchronously. By doing so, it enhances the overall system performance and enables the application throughput to scale linearly as a function of the number of cache servers. The main limitations of this technique are its increased software complexity and additional memory requirement.

When the data store is unavailable, a write-back policy buffers writes as long as the caching layer is available. Once the data store becomes available, it applies the buffered writes to the data store asynchronously. We use this idea to introduce TARDIS, a family of techniques to process writes when the data store is unavailable. TARDIS techniques (TAR, TARD, DIS and TARDIS) differ in how they apply buffered writes to the data store during the recovery mode. While TAR and DIS are simple to implement, other techniques are more complex. TARDIS is the most complex one and resembles the write-back policy.

We quantify the tradeoffs associated with the write-back policy and TARDIS family of techniques using YCSB, BG and TPC-C benchmarks. We compare their performance with alternatives such as write-around and write-through policies that perform writes synchronously. All benchmark results show that asynchronous writes enhance the overall performance and enable the application to scale as a function of the number of nodes in the caching layer. The results also highlight the extra memory required by the proposed techniques to buffer the writes.

Acknowledgment

I am most thankful for Professor Shahram Ghandeharizadeh's dedication and guidance that helped me through my Ph.D. journey. Professor Ghandeharizadeh has been more than an advisor to me, he is also my mentor and an exemplary role model. I was so fortunate to have the opportunity to work with him and to witness his devotion to his researches and to his students. I cannot thank him enough for his time and continuous encouragement in my Ph.D. research.

I thank the members of my qualification and thesis defense committee: Professor Clifford Neuman, Professor François Bar, Professor Wyatt Loyd and Professor Cyrus Shahabi for their guidance and constructive feedback. I thank the University of Southern California for providing us with the resources to carry out our experiments and researches.

I am grateful to my family, including my grandmother, my parents and my brother for believing in me. They encouraged and supported me through every challenge of my life.

I really enjoyed the moments with my friends and fellow researchers at the USC Database Laboratory: Jason Yap, Yazeed Alabdulkarim, Haoyu Huang and Marwan Almaymoni. They were my critics, my partners and my sources of motivation.

A very special gratitude goes out to the Vietnam Education Foundation (VEF). VEF has provided me with the initial fund and the first foundation to make my Ph.D. research in the US possible.

I show my deep gratitude to the Flux Research Group. I used their emulab nodes extensively in conducting experiments. Emulab enables a stable, isolated environment where experiments can be setup quickly and run consistently. Without emulab, it would be more challenging to complete the experiments provided in this dissertation.

Last but not least, I am thankful for my dearest girlfriend, Lien Cao, for being part of my life. Her support and inspiration enabled me to complete this dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Cache-Augmented Data Stores

The Cache Augmented Data Store (CADS) architecture extends a persistent data store with a caching layer using off-the-shelf commodity servers. It has been widely deployed in systems with workloads that exhibit a high read to write ratio. It is in use by social networking sites such as Facebook, Twitter and LinkedIn because it enhances system throughput and response time significantly [29, 30, 4, 58].

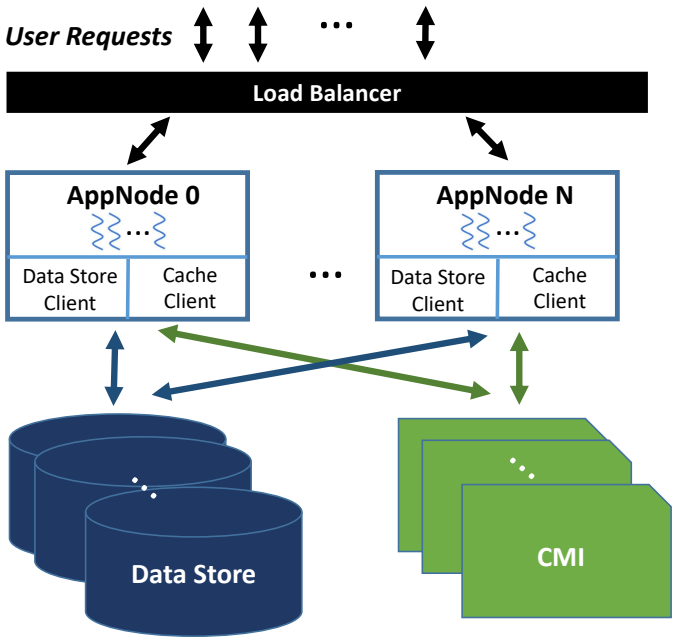
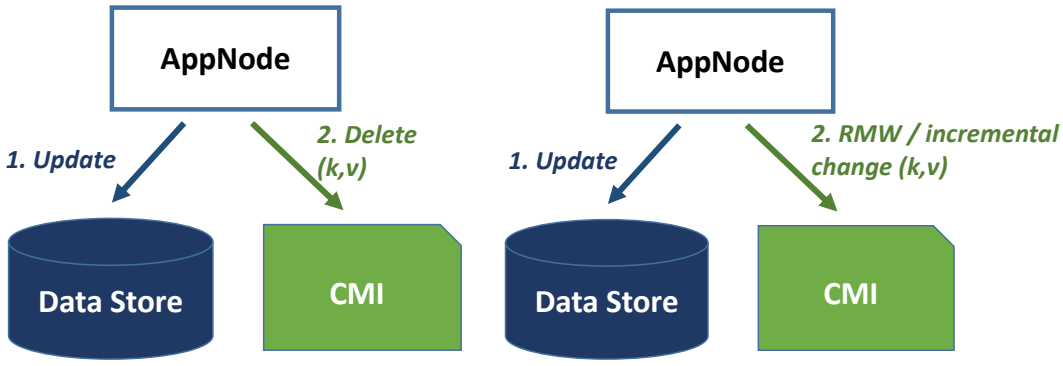


Figure 1.1: Architecture of a Cache Augmented Data Store.



(a) Write-around (b) Write-through

Figure 1.2: Write-around and write-through Policies.

Figure 1.1 shows the CADS architecture. A load balancer directs user requests to different application servers. Each application server consists of one or many AppNode instances serving user requests. Each AppNode has client components to communicate with the persistent data store (e.g., JDBC with SQL systems) and the cache (e.g., Whalin Client with memcached). The data store may either be a SQL (e.g., MySQL [56], PostgreSQL, OracleDB) or a NoSQL data store (e.g., MongoDB, CouchBase). Multiple cache manager instances (CMIs) may be deployed on a cache server. Example CMI includes an in-memory key-value store such as *memcached* [9] or *Redis* [64]. AppNodes communicate with CMIs via message passing. These key-value stores provide simple interfaces such as `get`, `set`, `incr`, `append` and `delete`.

The CADS architecture assumes that a software developer provides application logic to identify cached keys and how their values are computed using the data store. With a read, an AppNode identifies the key and looks up its value from the cache. If the value exists (cache hit), processing of the read completes using it. Otherwise, the value is missing from the cache (cache miss). The AppNode queries the data store for the data item (for example, a row with tabular database or a document with a document store) and computes the key-value pair to be

inserted as an entry in the cache for future reference. Reads benefit from the CADS architecture because result look-up from in-memory cache is much faster than query processing using data store [29, 58].

A write may be an insert, delete or update that changes data store state. It must maintain the cache entries consistent with data items from the data store. Figure 1.2 shows the write-around and write-through policies. With write-around (Invalidate), an AppNode deletes the cache entries corresponding to the updated data item. With write-through (Refill), it may employ incremental update (e.g., append or increment) or read-modify-write techniques to refresh the impacted cache entries. Both policies use synchronous writes, i.e, the AppNode issues the write request to the data store and waits for its acknowledgement.

Instead of executing writes synchronously, a write may be executed asynchronously by buffering its changes in the cache and returning success to the requester. These changes are applied later by a different action, thread or process. This technique has certain benefits. It allows a write to be executed even when the data store is unavailable, enhancing system availability. It scales the system throughput with the number of cache servers. It reduces the load on the data store. On the other side, it has several drawbacks. It increases software complexity, requiring more efforts for developers to develop and maintain the system. It also requires extra cache servers' memory.

1.2 Thesis Contributions

The thesis explores two applications of processing writes asynchronously in a CADS architecture.

First, we design and implement a write-back policy for CADS. Write-around and write-through limit performance and the scalability of the caching layer by performing writes synchronously. The write-back policy addresses this limitation by processing writes asynchronously: AppNodes buffer writes in the cache in forms of buffered writes and latterly apply them to the data store by using either background worker threads or reads observing cache misses. We present the key elements in designing and implementing a write-back policy that provides read-after-write consistency. Experimental results show the write-back policy enables the caching layer to dictate overall system performance and scale horizontally. With some workloads, it outperforms write-around and write-through policies by up to 50x. We also compare the proposed technique with write-back policy of other caches (e.g., host-side caches such as FlashCache) to demonstrate its superiority.

Second, we propose TARDIS, a family of techniques that process write asynchronously in the presence of data store unavailability. By processing a write, it must be durable, i.e, its changes persist once the AppNode confirms the operation as success. Subsequent reads must observe the changes even if there are failures in the system. Durable writes are important because they support consistency. When there are failures in the system, providing durable writes is challenging. The goal of TARDIS is to provide an always-writable experience for CADS in the presence of failed writes to the data store.

1.3 Reader's Guide

Chapter 2 presents the work related to the research. Chapter 3 introduces designs and implementations of a write-back policy for CADS. Chapter 4 describes TARDIS, a family of techniques to enhance system availability when the data store

is unavailable. Chapter 5 presents a cache framework supporting write-back policy. Chapter 6 proposes the future plan to extend the completed works.

Chapter 2

Related Work

2.1 Consistency

The CAP theorem states a system designer must choose between strong consistency and availability in the presence of network partitions [15, 51]. TARDIS improves the availability of the system by teleporting writes when the data store fails. TARDIS retains the same consistency as the data store as long as buffered writes are not lost.

A weak form of data consistency known as eventual has multiple meanings in distributed systems [72]. In the context of a system with multiple replicas of a data item, this form of consistency implies writes to one replica will eventually apply to other replicas, and if all replicas receive the same set of writes, they will have the same values for all data. Historically, it renders data available for reads and writes in the presence of network partitions that separate different copies of a data item from one another and cause them to diverge [21, 73, 12]. TARDIS handles network partitions that cause failed writes. It also teleports failed writes due to the failure of the data store.

The race conditions encountered during recovery (see Section 4.4.1) are similar to those in geo-replicated data distributed across multiple data centers. Techniques such as causal consistency [48, 49, 46] and lazy replication [45] mark writes with their causal dependencies. They wait for those dependencies to be satisfied prior to applying them at a replica, preserving order across two causal writes. TARDIS is

different because it uses Δ_i to maintain the order of writes for a document D_i that observes a cache miss and is referenced by one or more failed writes. With cache hits, the latest value of the keys reflects the order in which writes were performed. In recovery mode, TARDIS requires the AppNode to perform a read or a write action by either using the latest value of keys (cache hit) or Δ_i (cache miss) to restore the document in the data store prior to processing the action. Moreover, it employs AR workers to propagate writes to persistent store during recovery. It uses leases to coordinate AppNode and AR workers because its assumed data center setting provides a low latency network.

Neumerous studies perform writes with a mobile device that caches data from a database (file) server. (See [70, 61] as two examples.) Similar to TARDIS , these studies enable a write while the mobile device is disconnected from its shared persistent store. However, their architecture is different, making their design decisions inappropriate for our use and vice versa. These assume a mobile device implements an application with a local cache, i.e., AppNode and the cache are in one mobile device. In our environment, the cache is shared among multiple AppNodes and a write performed by one AppNode is visible to a different AppNode - this is not true with multiple mobile devices. Hence, we must use leases to detect and prevent undesirable race conditions between multiple AppNode threads issuing read and write actions to the cache, providing correctness as long as buffered writes are not lost. Finally, mobile devices may differentiate buffered writes from other data competing for cache space, minimizing loss. This is different than our assumed environment where buffered write compete with other data for cache space.

2.2 Host-side Caches

Host-side Caches [5, 44, 17, 38, 34] (HsC) such as Flashcache [52] and bcache [67] are application *transparent* caches that stage the frequently referenced disk pages onto NAND flash. These caches may be configured with either write-around, write-through, or write-back policy. They are an intermediary between a data store issuing read and write of blocks to devices managed by the operating system (OS). Application caches such as memcached are different than HsC because they require application specific software to maintain cached key-value pairs. TARDIS is somewhat similar to the write-back policy of HsC because it buffers writes in cache and propogates them to the data store (HsC's disk) in the background. TARDIS is different because it applies when writes to the data store fail. Elements of TARDIS can be used to implement write-back policy with caches such as memcached, Redis, Google Guava [32], Apache Ignite [8], KOSAR [27], and others.

2.3 Recovery Techniques

Two main approaches for recovery are *write-ahead logging* [53] and *shadow paging* [37]. Write-ahead logging (WAL) writes all modifications to a log before applying them to the database. It then applies the changes to the disk pages in-place with the help of background workers. An advantage of WAL is that writing log records to disk sequentially is much faster than writing randomly. When a database recovers after failure, it applies **Redo** with committed transactions and **Undo** with uncommitted ones. Different than WAL, shadow paging applies the

changes to an allocated page (shadow page). Since shadow page is not referenced anywhere other than this transaction, isolation and atomicity are preserved.

Deferred and immediate database modification are two common techniques in log-based recovery. Deferred database modification defers writing log records to disk until the transaction commits. At commit point, all logged operations of the transaction are written to the disk and a redo is applied to log records. If the transaction rolls back, it simply discards any corresponding records in the log. Immediate database modification updates the database in-place as the transaction goes. If the transaction rolls back, it needs to undo all the changes made since the transaction starts to ensures atomicity.

SnapshotRecovery algorithms implemented in TARDIS are somewhat similar to deferred database modification. Prior to updating the data store, it maintains the updates in the cache and deletes them when the update to the data store is acknowledged. The differences between our techniques and deferred database modification are: 1) the cached data item could not be discarded if the update fails, and 2) With SnapshotRecovery, we update the cached data item in isolation with new writes. In the presence of AppNode failure, new writes are buffered to a new buffered writes key-value pair P with a different uid that isolates with the log created by the previously AppNode failure. New buffered writes are not processed until the failure is recovered. Hence, it is somewhat similar to shadow paging.

2.4 Cache Middleware

Caching middleware such as Oracle Coherence [60], EhCache [69], Infinispan [42] or IBM WebSphere eXtreme Scale [41] support write-back policy. Coherence, Ignite, and EhCache are similar and we describe them collectively using Coherence's terminology.

Coherence provides a simple "put(key,value)" that (1) inserts the key-value pair in the cache, overwriting it if it exists, and (2) places the key-value pair in a CacheStore queue before returning success. Coherence is data store agnostic by requiring the developer to implement the "store()" interface of CacheStore. After a configurable time interval, a background thread invokes store() to persist the (key,value) pair to the data store. A read (issued using get(key)) either observes the latest value from the cache or requires the cache to load the missing cache entry. This is realized by requiring the developer to implement the "load()" interface that queries the data store for the missing key-value pair. While the Coherence documentation is not specific about the details of how cache misses are processed, we speculate their processing considers the queued writes to provide read-after-write consistency.

Our proposed write-back is different in several ways. First, the concept of a buffered write is absent with Coherence, Ignite and EhCache. One may argue it is the cache entry inserted in a queue for asynchronous processing. With our technique, a buffered write is explicit. It may contain arbitrary changes that are independent of the cached entry. To illustrate, with MongoDB, the queued entry of Coherence is a document equivalent to the one stored in the cache. If an update modifies one attribute of a YCSB document then the put method writes

the entire[1] document to cache which in turn causes the "store()" implementation to write the entire document to MongoDB. With our technique, a buffered write may maintain the document id, the property referenced by the update, and its new value. The developer would implement the interface to update the property with the new value for the specified document id in MongoDB. This flexibility requires our technique to provide the concept of mapping. Coherence does not require this concept because its queued entry is the same as the cache entry.

IBM Websphere cache may be divided into maps, where each map is a collection of cache entries. An application may configure a loader (similar to CacheStore interfaces of Coherence) for a map. With write-back, it creates a thread to process delegating requests coming to a loader. When a write inserts, updates or deletes an entry from a map, a LogElement object is generated and queued. Each LogElement object records the operation type (insert, update or delete) and the new and old values. Each map has its own queue. A write-behind thread is initialized to periodically remove a queue element and apply it to the data store. LogElement objects are similar to our buffered writes using Append approach. However, the documentation does not describe how to provide durability for LogElement objects. Moreover, it does not describe how to process cache misses with pending LogElement objects in the queue. Hence, it lacks mapping as a concept.

2.5 Everest

Similar to TARDIS, Everest [57] uses "partial write-back". It is designed to improve the performance of overloaded volumes. Each Everest client has a base volume and a store set. When the base volume is overloaded, the client off-loads

[1]This may require the update to query MongoDB for the entire document in order to compute the new version of the document with the update.

the writes to the idle stores. When the base volume load fall below a threshold, the client uses background threads to apply writes to base volume. TARDIS buffers writes in the cache when the data store is unavailable (either because the data store server goes down or there is network connectivity issue between the application server and the data store). Subsequently, when the data store is available, TARDIS enters the recovery mode. During this mode, the application server retrieves the buffered writes from the cache and applies them to the data store. With our write-back design, background worker threads apply the buffered writes every time they are present in the cache. This is different than partial write-back technique where worker threads apply buffered writes during recovery mode only.

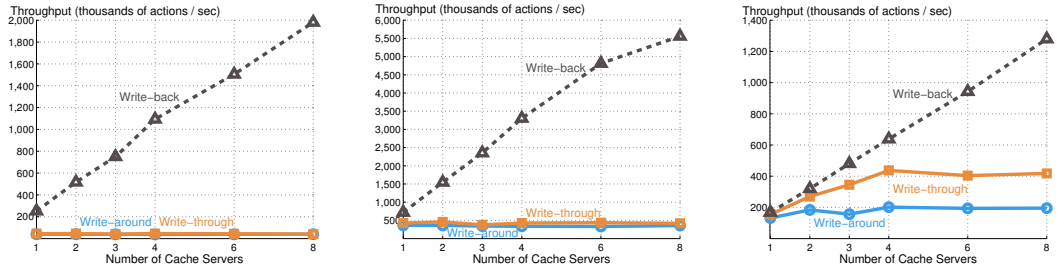
Chapter 3

Write-back Policy

Write-around (invalidation) and write-through (refill) policies apply a write to the data store synchronously, see Section 1.1. This prevents the caching tier from absorbing writes, requiring the data store layer to scale to process writes even when its extra capacity is not required for read load. This chapter presents a client-side implementation of write-back (write-behind) policy to address this limitation. The proposed technique buffers writes in the cache and applies them to the data store asynchronously.

Write-back enhances both the performance and horizontal scalability of a CADS system significantly. To illustrate, Figure 3.1 shows the scalability of a CADS configuration consisting of one MongoDB server as we vary the number of servers in its caching layer from one to eight. We show results for several Yahoo! Cloud Services Benchmark [19] (YCSB) workloads: the write heavy Workload A and read heavy Workloads B and S. With write-around and write-through, the caching layer does not scale because the data store is the bottleneck. With write-back, the throughput scales almost linearly even though the data store remains fully utilized. This is because write-back buffers writes in the caching layer and applies them to the data store asynchronously, removing the data store from the critical path of processing requests.

These results explain why caching middleware such as Oracle Coherence [60], EhCache [69] and Infinispan [42] support the write-back (*write-behind*) policy. They do so by providing simple interfaces of a key-value store such as get, put



(a) Workload A (b) Workload B (c) Workload S

Figure 3.1: Throughput of different YCSB workloads with write-around, write-through and write-back policies.

and delete. A developer is responsible for providing an application specific implementation of these interfaces.

Design and implementation of a write-back policy must address several challenges. First, how to represent data store writes (termed *buffered writes*) as cache entries and how to prevent them from being evicted by the cache replacement policy. Second, how to apply the buffered writes from the cache to the data store efficiently and ensure read-after-write consistency. Reads must always observe values produced by the latest writes even when they are buffered in the cache. Otherwise, they may produce stale results that impact the correctness of application and pollute the cache. Third, how to provide durability of writes in the presence of cache failures. If a write is acknowledged and its buffered writes are lost due to a cache server failure then the write is no longer durable. Fourth, how to process a non-idempotent buffered write such as increment without compromising consistency in the presence of failures.

This study presents a write-back technique that addresses the above challenges. Our proposed technique provides:

- **Read-after-write consistency.** A read is guaranteed to observe the latest writes. A read that references a cache entry with pending buffered writes

is processed in the context of these writes. We also use leases to prevent undesirable race conditions.

- **High performance and scalability.** Our implementation of write-back scales the system throughput as we increase the number of cache servers. This is achieved by partitioning buffered writes across cache servers and using the client component of the caches to apply these writes to the data store asynchronously.

- **Durability.** Buffered writes are pinned in memory, preventing their eviction by the cache replacement policy. Moreover, we replicate buffered writes across 3 or more cache servers to tolerate cache server failures. These replicas may be assigned to servers in different racks within a data center. To tolerate data center failure, one may use non-volatile memory such as today's NVDIMM-N [66].

Our implementation uses leases (see Section 3.1) to provide read and write atomicity at the granularity of a one or more cache entries and a single data store transaction. Correctness of a write is the responsibility of the application developer. It should transition the database from one consistent state to another [36]. (Transaction processing systems [36] make the same assumption.) Key elements of our design include:

1. To use a write-back policy, an application must provide a *mapping* of cache entries to buffered writes. This is because a cache miss by a read must identify those pending writes that impact its result, apply them to the data store, and then query the data store for the missing cache value. Without this mapping, the application may not provide read-after-write consistency.

2. Use leases to provide read-after-write consistency and enhance data availability in the presence of failures.

3. Store buffered writes and their mappings in the caching layer by pinning them. With atomic operations (termed sessions) that impact more than one key-value pair, we capture their spatial relationship in the buffered writes. Similarly, we capture the temporal relationship between sessions that impact the same key-value pairs to apply buffered writes to the data store in the same order.

4. Differentiate between idempotent and non-idempotent changes. We present two different techniques to support non-idempotent writes with NoSQL and SQL systems.

5. Replicate buffered writes across multiple cache servers to enhance availability in the presence of failures.

6. Partition buffered writes to load balance across multiple cache servers and use client component of the cache to minimize server software complexity (load) and prevent single source of bottleneck.

While all caching middleware advocate partitioning of buffered writes for scalability and their replication for high availability as best practices [60, 69, 42, 41], they lack the first four aforementioned design elements: mappings, leases, spatial and temporal relationships of buffered writes, and support for non-idempotent changes. These differences make it challenging to compare our technique with the existing caching middleware, see Chapter 2.

This paper makes several *contributions*. First, we introduce the concept of mapping and use of leases to provide read-after-write consistency with cache misses.

Second, we support non-idempotent changes with both No-SQL and SQL systems. Third, we present an implementation of these concepts in two different designs. While Design 1 is appropriate for a document (relational) store that provides ACID properties at the granularity of one document (row), Design 2 supports complex transactions consisting of multiple statements impacting multiple documents (rows) or cache entries. Fourth, we provide a comprehensive evaluation of a client-side implementation of write-back using off-the-shelf software components with YCSB [19], BG [13], and TPC-C [71] benchmarks. Obtained results show write-back enhances performance with all benchmarks as long as memory is abundant, out-performing both write-through and write-around polices.

We also evaluate the impact of the following factors on write-back performance: the number of background threads (BGTs), maximum amount of pinned memory for buffered writes, and degree of replication for buffered writes. Increasing the number of BGTs reduces the amount of memory required by write-back. However, it also reduces the throughput observed by the application.

We compare the write-back policy with those used in host-side caches and caches of data stores such as MongoDB, i.e., MongoDB configured with writeConcern set to ACKNOWLEDGED. The proposed write-back policy complements its block-based alternatives, host-side caches [17, 22, 39, 44] and MongoDB's write-back technique, enhancing their performance several folds.

Write-back has two limitations. First, it is more complex to implement than either write-through or write-around policies, requiring additional software. Second, its performance with a limited amount of memory may be inferior when compared with the other policies. Specifically, we observe calcification [43] of memory with memcached when the size of mappings and buffered writes is different. This prevents write-back from utilizing its maximum allocated pinned memory.

The rest of this chapter is organized as follows. Section 3.2 describes the design of the write-back policy. Section 3.3 evaluates the write-back policy by comparing it with other write policies and use of write-back with other caches such as host-side cache. Section 3.4 provides a formal proof of providing read-after-write consistency for write-back.

3.1 Overview

Definition 3.1.1. *A cache entry is represented as a key-value pair (k_i, v_i) where k_i identifies the entry and v_i is the value of the entry. Both k_i and v_i are application specific and authored by a developer.*

An AppNode read identifies the key k_i and gets its value v_i from the cache. If the value v_i exists then the read has observed a cache hit and proceeds to consume v_i. Otherwise, with a cache miss, the read queries the data store for the data item. It may fetch a few rows of a relational database or a document of a document store to compute the key-value pair to be inserted in the cache for future reference. Reads that observe a hit for these key-value pairs benefit because result look-up from the in-memory cache is much faster than query processing using the data store [29, 58].

Figure 1.2 shows the write-around (invalidate) and write-through (refill) policies. Both update the data store synchronously. While write-around deletes the impacted key-value pairs, write-through updates them. Write-through may employ incremental update (e.g., append or increment) or read-modify-write to update a key-value pair. With write-back, see Figure 3.2, a write updates the impacted key-value pairs similar to write-through. However, it stores one or more replicas of its changes (termed *buffered write*) in the CMI instead of applying it to the

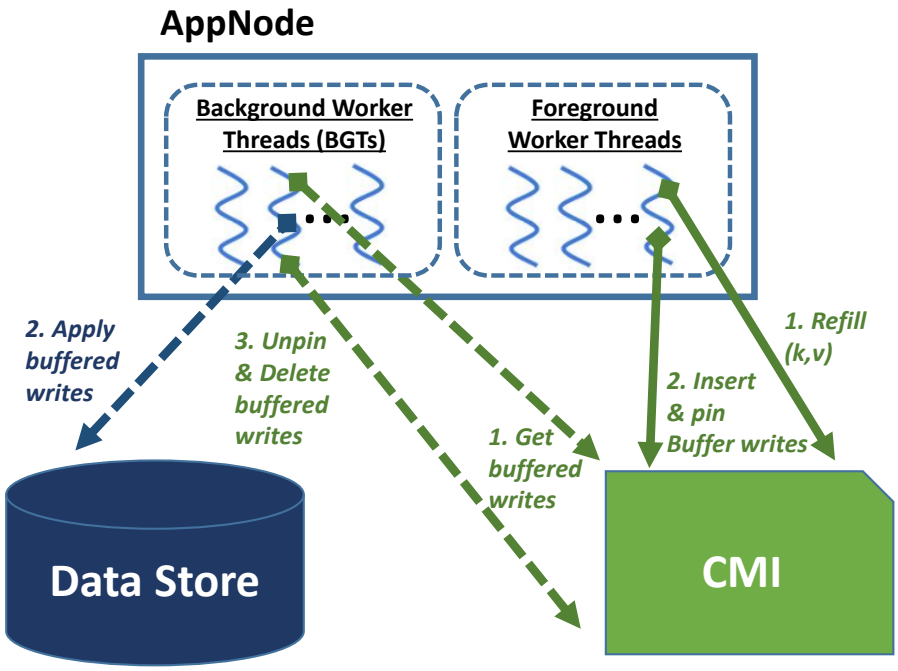


Figure 3.2: Write-back policy. Solid arrows are performed in the foreground. Dashed arrows are performed in the background.

data store. The write is then acknowledged to the user as successful. Background threads, BGTs, apply the buffered write to the data store asynchronously. Figure 3.2 shows these threads are co-located with AppNode. However, this is not a requirement and a different process may host these threads.

We make several *assumptions* about the cache manager that may not be standard. First, the application may *pin* and *un-pin* a key-value pair when setting it in a CMI. This means the CMI's cache replacement technique may not evict a pinned key-value pair. We pin buffered writes and their mappings in one or more CMIs. A background thread that applies a buffered write to the data store un-pins and deletes them.

Second, we assume the concept of sessions. A session is an atomic operation with a unique identifier. It reads and writes one or more cache entries and issues one transaction to the data store. We use a session to implement a transaction

of the TPC-C benchmark. A session that reads a key-value pair must obtain a Shared (S) lease on it prior to reading it. A session that writes a key-value pair must obtain an eXclusive (X) lease on it prior to updating[1] its value.

Requested Lease	Existing Lease	
	S	X
S	Grant S lease	Abort and Retry
X	Grant X and void S lease	Abort and Retry

Table 3.1: S and X Lease Compatibility.

The S and X leases are different than read and write locks in several ways. First, S and X leases are non-blocking. As shown in the compatibility Table 3.1, when a session T_r requests a S lease on a key-value pair with an existing X lease, it aborts and retries. Second, when a session T_r requests an X lease on a data item with an existing S lease granted to session T_h, T_r wounds T_h by voiding its S lease. At its subsequent request, T_h is notified to abort and restart. This prevents write sessions from starving. Third, they have a finite lifetime in the order of hundreds of milliseconds. Once they expire, their referenced key-value pair is deleted. This is suitable for a distributed environment where an application node fails causing its sessions holding leases to be lost. Leases of these sessions expire after sometime to make their referenced data items available again.

Prior to committing its data store transaction, a session validates itself to ensure all its leases are valid. Once a session is validated, its S leases become golden. This means they may no longer be voided by an X lease. An X lease that encounters a golden S lease [10] is forced to abort and retry. Once the session commits its database transaction, it commits the session and releases its leases.

[1]An update may be in the form of a read-modify-write or incremental update such as increment, append, etc.

Different components of a distributed CADS may implement the write-back policy and its buffered writes. For example, it may be implemented by the application instances, CMIs, or a middleware between the application instances and CMIs, or a hybrid of these. This paper describes an implementation using the application instances, see Figure 3.2.

3.2 Write-Back Policy: Two Designs

This section presents two different designs for the write-back policy. They assume the data store transaction that constitutes a session has different complexity. Design 1 assumes simple data store transactions that either read or write a single document (row) of a document (relational) store such as MongoDB (MySQL). It is suitable for workloads modeled by the YCSB benchmark. Design 2 assumes a session's data store transaction reads and/or writes multiple rows of a SQL (or documents of a transactional MongoDB) data store. It is suitable for complex workloads such as those modeled by the TPC-C benchmark.

Design 1 maintains changes at the granularity of each document. A cache look up that observes a miss is provided with a *mapping* that identifies changes that should be applied to the data store prior to querying the data store for the missing cache entry. To enable BGTs to discover and apply changes to the data store asynchronously, it maintains a list of documents with changes, termed PendingWrites.

Design 2 maintains changes at the granularity of a session. Its mapping identifies the dependence of a cache entry on the sessions with pending changes. A cache miss uses this mapping to identify sessions with pending buffered writes and apply their changes to the data store. Next, it queries the database for the missing cache entry. It maintains a queue that specifies the order in which sessions commit. A

BGT uses this queue to discover sessions with pending writes and applies them to the data store in the same order as their serial commit.

Design 2 is different than Design 1 in that its queue identifies temporal dependence of the sessions across multiple data items in the data store. Design 1 is simpler because it maintains the order of changes per document by assuming a session writes only one document. Design 2 is essential for preserving SQL's integrity constraints such as foreign key dependencies between rows of different tables.

Below, we provide a formal definition of a change, a buffered write, and a mapping. Subsequently, we describe PendingWrites and queues to facilitate discovery of buffered writes by BGTs. Finally, we present BGTs and how they apply buffered writes to the data store. Each discussion presents the two designs in turn.

A Change: A change is created by a write session. It may be idempotent or non-idempotent. Both designs must apply a non-idempotent write to the data store once. This is specially true with arbitrary failures of AppNodes. The definition of a change is specific to the data store's data model. It is different for the document data model (NoSQL) of Design 1 when compared with the relational data model (SQL) of Design 2. Below, we describe these in turn. Subsequently, we describe how each design supports non-idempotent changes.

With Design 1, a change by a write may be represented in a *JSON-like* format that is similar to MongoDB's update command. Table 3.2 shows examples of how these changes are represented. In this table, `$set` is idempotent, while `$inc` is non-idempotent. A change that adds a value or an object to a set while guaranteeing uniqueness (e.g., `$addToSet` of MongoDB) is idempotent since it does not allow duplicates. However, a similar operation without uniqueness property (MongoDB's `$push`) is non-idempotent.

Document D	Change applied to D	Idem-potent?	After applying the write to D
{ field: “val” }	Set value of *field* to *newval* { “$set”: { field: “newVal” } }	✓	{ field: “newVal” }
{ field: “val” }	Remove a *field* { “$unset”: { field: “” } }	✓	{ }
{ field: i }	Increment value of *field* by x { “$inc”: { field: x } }	✗	{ field: i+x }
{ field: [“a”] }	Add to value of *field* { “$addToSet”: { field: “b” } }	✓	{ field: [“a”, “b”] }
{ field: [“a”] }	Add to value of *field* { “$push”: { field: “a” } }	✗	{ field: [“a”, “a”] }
{ field: [“a”, “b”] }	Remove from value of *field* { “$pull”: { field: a } }	✗	{ field: [“b”] }

Table 3.2: Example of write actions and the representation of their changes.

With Design 2, a change may be a SQL DML command: insert, delete, update. The command may impact multiple rows. Design 2 may represent the change as a string representation of the DML command issued by the application or a compact representation of it. In our TPC-C implementation, we use the latter to enhance utilization of both memory space and network bandwidth.

Designs 1 and 2 process non-idempotent changes in different ways to tolerate arbitrary failures of BGTs that apply these changes. Design 1 requires developers to provide additional software to convert non-idempotent changes to idempotent ones. BGTs replace a non-idempotent change with its equivalent idempotent change prior to applying it to the data store. Design 2 uses the transactional property of its data stores to apply non-idempotent changes only once. Its BGTs stores the id of a session applied to the data store in a special table/collection, *AppliedSessions*, as a part of the transaction that applies this session's changes to the data store. Prior to applying changes of a session to the data store, a BGT looks up the session id in the AppliedSessions table. If found then it discards the session and its buffered writes. Otherwise, it constructs a transaction consisting of the session's changes (SQL DML commands) along with the command that appends the id of the session to

the AppliedSessions table. Next it executes this transaction and deletes the session object from the cache. Periodically, a BGT compacts the AppliedSessions table by deleting those session rows with no session objects in the CMI.

Buffered writes: A buffered write is a sequence of changes. With Design 1, atomicity is at the granularity of a document [54]. Hence, a buffered write represents a sequence of changes to one document. Each is a represented as a pinned cache entry, i.e., a key-value pair in a CMI. Buffered writes are partitioned across CMIs.

Definition 3.2.1. *With Design 1, a buffered write for a document D_i is represented as a key-value pair (k_i^{bw}, v_i^{bw}) where k_i^{bw} identifies a buffered write and v_i^{bw} stores either the final value of D_i or the pending changes to D_i. With the latter, the sequence of changes in v_i^{bw} represents the serial order of writes to D_i.*

The key k_i^{bw} may be constructed by concatenating "BW" with DocID where DocID is a unique identifier (or primary key) of the impacted document.

There are two approaches to buffer a change to v_i^{bw}: Append and Read-Modify-Write (RMW). Both acquire an X lease on the key k_i^{bw}. While Append requires the AppNode to append its changes to v_i^{bw}, RMW requires the AppNode to read v_i^{bw}, update v_i^{bw} with the change, and write v_i^{bw} back to the cache. An efficient design of RMW grants an X lease as a part of read that fetches v_i^{bw}. RMW may compact v_i^{bw} by eliminating changes that nullify one another. Below, we present an example to illustrate these concepts.

Example 3.2.1. *Alice's document is impacted by two write actions: i) Bob invites Alice to be friend and ii) Alice accepts Bob's invitation. Representation of changes for i) is { "$addToSet": { pendingfriends: "Bob" } }, i.e., add Bob to Alice's pending friends. Representation of changes for ii) is { "$pull": { pendingfriends:*

"Bob" }, "$addToSet": { friends: "Bob" } }, i.e., remove Bob from pending friend list and add Bob to Alice's friend list. With Append, the buffered write is merely the JSON array that includes the two representations. With RMW, the buffered write becomes { "$addToSet": { friends: "Bob" } } since $pull cancels $addToSet of Bob to pendingfriends of Alice. ■

In Example 3.2.1, both write actions make small changes to Alice's document. With append, the changes reflect the serial order of writes. The RMW performs a compaction to remove a redundant change.

Design 2 must support sessions that produce changes impacting rows of multiple tables. Thus, a buffered write represents a sequence of changes performed by a session. It is associated with a session object.

Definition 3.2.2. *With Design 2, a buffered write for a session T_i is represented as a key-value pair (k_i^{bw}, v_i^{bw}) where k_i^{bw} identifies the session T_i and v_i^{bw} stores either the raw SQL DML commands issued by that session or their compact representation. The sequence of changes in v_i^{bw} represents the serial order of SQL DML commands by session T_i.*

With both designs, buffered writes may be replicated across multiple CMIs to enhance their availability in the presence of CMI failures. These replicas are un-pinned and deleted after they are applied to the data store.

Mapping: A mapping is a detective technique that provides read-after-write consistency when the application encounters misses for cache entries with pending buffered writes. The application applies these buffered writes to the data store and then query the data store to compute the missing values. A mapping enables a cache miss to discover writes that must be applied to the data store.

An alternative to the detective approach of mappings is to prevent cache misses for entries with pending buffered writes. The idea is to require a write session to generate the missing cache entry prior to generating a buffered write (to prevent a future cache miss). This solution must pin these cache entries to prevent their eviction until their buffered writes are applied to the data store. The challenge with this design is that it may pin entries that may not be referenced in the near future, reducing the cache hit rate of the application. Due to this limitation, we discard this preventive technique and assume use of mappings for the rest of this paper.

There are two ways to use a mapping. First, apply buffered writes prior to evicting a cache entry that depends on them. We term this the cache-server side (CSS) solution. Second, require the reads that observe a cache miss to look up the mapping to identify buffered writes, apply them to the data store first, and query the data store to compute the missing value. We term this technique the application-side solution (APS).

With both APS and CSS, it is the responsibility of the developer to author software to specify mappings.

Definition 3.2.3. *A* mapping *inputs the key k_i for a cache entry to compute keys of its buffered writes, $\{k_i^{bw}\}$. With Design 1, these keys identify the documents with buffered writes that are used to compute the missing entry. With Design 2, these keys identify the session ids with pending buffered writes. A mapping may be a developer provided function or represented as a key-value pair (k_i^M, v_i^M).*

When a mapping is represented as a pinned key-value pair (k_i^M, v_i^M), k_i^M identifies mapping i uniquely. With Design 1 (2), its value v_i^M is the keys of those buffered writes (session objects) that impact k_i. A write that generates buffered writes must also generate a mapping. A read that observes a cache miss must

always look-up $\{k_i^M\}$ in the cache and apply its identified buffered writes to the data store prior to querying it for the missing value. Many mappings for different cache entries may reside in a CMI.

It is possible for the developer to specify a mapping using an order preserving data structure such as B-tree or an interval tree. This applies when a write impacts a range of values. In this case, a read that references a missing cache entry must look up the overlapping range to identify the relevant buffered writes [65].

Algorithm 1: Process a cache miss (Design 1).

```
1 function Process_Cache_Miss(k_i):
      // Use mapping to compute key of the buffered write
2     k_i^bw ← mapping(k_i);
3     if k_i^bw is null then
4         return; // Cache miss, no buffered write
5     Apply_Buffered_Write(k_i^bw);
6     Un-pin and delete explicit mappings (if any);
7     v_i ← Result of a function that queries the data store;
8     Insert (k_i, v_i) in CMI[hash(k_i)];
```

With APS, a read that observes a cache miss uses output of a mapping to look up the buffered write(s) in the CMI, see ② of Algorithm 1. If there is no mapping then it returns a cache miss so that the application may query the data store for the missing value ③. Otherwise, it applies the buffered writes to the data store prior to computing the missing cache entry, see Algorithm 2. This algorithm applies a buffered write as an atomic session because, with Design 1, it must convert non-idempotent changes into idempotent ones. Algorithm 2's read of a buffered write for RMW obtains an X lease and fetches the buffered write ③. Next, it compacts the changes in this buffered write ④. If the buffered write contains one or more non-idempotent changes, it converts these changes to idempotent ones, writing a revised buffered write with idempotent changes only, ⑤-⑧. It calls itself

recursively ⑨ to apply the idempotent writes to the data store, unpins and deletes the buffered write, and commits to release its X lease, ⑪-⑬.

Algorithm 2: Apply buffered write (Design 1).

```
1  function Apply_Buffered_Write(k_i^bw):
       // Look up buffered write
2      sessionId ← Generate a unique token;
3      v_i^bw ← Read(k_i^bw, sessionId); // Obtain a X lease on the buffered write
4      Compact v_i^bw and remove redundant changes;
5      if v_i^bw contains one non-idempotent change then
6          v_i^bw ← Idempotent equivalent of v_i^bw;
           // Replace value of buffered write with its idempotent equivalent
7          Set k_i^bw to v_i^bw;
8          Commit(sessionId); // Release X lease
9          Apply_Buffered_Write(k_i^bw);
10     else
11         Apply v_i^bw to document D_i in the data store;
           // Delete the buffered write
12         Un-pin and delete k_i^bw;
13         Commit(sessionId);
```

Similarly, CSS uses the output of the same mapping to locate buffered writes for a key that is being evicted, applies them (if any) to the data store per Algorithm 2. This technique incurs the overhead of an extra cache look up for every cache miss with APS and cache eviction with CSS.

APS and CSS differ in how they look up the mappings and process them. CSS uses the mappings when evicting a cache entry. APS uses the mappings when a read observes a cache miss. APS and CSS look up the mappings at the same rate with a workload that has the same rate of cache misses and evictions. If the cache replacement technique favors maintaining small sized cache entries by evicting large ones [28] then its rate of cache evictions would be lower than its cache misses. In this case CSS would do fewer look ups of mappings and processing of buffered writes.

There is also the complexity of implementing APS and CSS in a distributed manner. With an off-the-shelf cache manager such as memcached, it may be easier to implement APS instead of CSS. Without loss of generality, we assume APS for the rest of this paper.

Both Designs 1 and 2 may generate a Query Result Change, QRC. As suggested by its name, a QRC impacts the results of a query. Algorithm 3 shows this pseudo-code. With a cache miss, the system may query the data store ② and apply QRCs to its results to obtain the latest value ③-⑦. The final results are inserted in the cache for future lookup. It is the responsibility of the application to maintain this entry up to date in the presence of writes that impact its value. Algorithm 3 does not delete QRCs. A BGT that applies a buffered write corresponding to one or more QRCs deletes them. This enables repeated applications of QRCs should the computed cache entry be evicted. We use Algorithm 3 to implement the Stock-Level transaction of TPC-C.

Discovery of buffered writes using PendingWrites and Queues: With both designs, a BGT must discover buffered writes and apply them to the data store. With Design 1, a special key termed *PendingWrites* identifies the buffered writes for different documents. A write session obtains an X lease on PendingWrites and appends its buffered write key (k_i^{bw}) to its value. To minimize contention, among concurrent writes we represent PendingWrites as α pinned sub-keys. Keys of the buffered writes (k_i^{bw}s) are hash partitioned across them. The value of α is typically a multiple of the number of concurrent writes (threads) and background threads, whichever is greater.

With Design 2, a queue maintains the serial order in which buffered writes of different sessions must be applied to the data store. It is represented as a key-value pair. The key identifies a queue known to a BGT and its value is a temporal order

of session objects to be applied to the data store. (Each session object identifies a buffered write.) An application may maintain multiple queues. For example, with TPC-C, there is one queue per warehouse. When TPC-C is configured with W warehouses, it maintains W queues.

Algorithm 3: Process a cache miss (Design 2).

```
1 function Process_Cache_Miss(k_i):
2     v_i ← Result of a function that queries the data store;
      // Use mapping to get a list of session ids
3     sessIds ← Get a list of session ids from mapping(k_i);
4     for each sessId in sessIds do
5         k_i^bw ← Construct the buffered write key from sessId;
6         v_i^bw ← Get buffered write k_i^bw;          // Obtain a S lease
7         Update v_i to include changes from v_i^bw;
8     Insert (k_i, v_i) in CMI;
```

Background Threads, BGTs: BGTs are threads that implement the asynchronous application of buffered writes to the data store. With Design 1, a BGT checks for dirty documents by looking up partitions of PendingWrites periodically. It processes each buffered write per Algorithm 2. This algorithm was presented in the context of processing a cache miss using a mapping.

Design 2 uses Algorithm 4 to apply a set of sessions to the data store. The set of sessions is obtained from a Queue. For each session, it looks up the id of the session in the AppliedSessions table ⑤. If it exists then the session has already been applied to the data store and is deleted. Otherwise, it creates a session and obtains the list of changes from each session object ⑪-⑬. It starts a data store transaction and inserts the id of the session in the AppliedSessions table, Step 15. It merges the list of changes and applies them to the data store ⑯. Finally, it commits the transaction ⑰. It maintains the id of sessions in a set named sessIdsToDelete ⑳. Once this set reaches a pre-specified threshold β, Algorithm 4

Algorithm 4: Apply sessions to the data store (Design 2).

```
1  function Apply_Buffered_Writes(sessIds):
2      sessIdsToApply ← {};
3      sessIdsToDelete ← {};
4      for each sessId in sessIds do
5          if sessId exists in Applied-Sessions table then
6              Delete buffered write of sessId from the cache;
7              Delete sessId from the Applied-Sessions table;
8              continue;
9          Add sessId to sessIdsToApply;
10         if the list sessIds is exhausted or sessIdsToApply size reaches α then
               // Apply sessions as one transaction to the data store
11             sessionId ← Generate a unique token;
12             {k_i^bw} ← Construct keys from sessIdsToApply ;        // Obtain X
                leases on k_i^bw
13             {v_i^bw} ← multi_read({k_i^bw}, sessionId);
14             Start data store transaction;
15             Insert sessIdsToApply to Applied-Sessions table;
16             Merge {v_i^bw} and apply them to the data store;
17             Commit data store transaction;
18             multi_delete({k_i^bw}, sessionId);      // Delete buffered writes of
                sessIds from the cache
19             Commit(sessionId);                       // Release X leases on k_i^bw
20             Add ids from sessIdsToApply to sessIdsToDelete;
21             sessIdsToApply ← {};                              // Clear the lists
               // Perform compaction
22             if the list sessIds is exhausted or sessIdsToDelete size reaches β
                then
23                 Delete sessIdsToDelete from the mappings;
24                 Delete sessIdsToDelete from Queue;
25                 Delete sessIdsToDelete from the Applied-Sessions table;
26                 sessIdsToDelete ← {};
```

deletes the session ids from the mappings, the Queues, and the AppliedSession table.

Durability: Both designs pin their buffered writes, mappings, PendingWrites/Queues in a CMI to prevent its replacement technique from evicting them. Moreover, with multiple CMIs, both designs replicate these entries to enhance their availability in the presence of CMI failures.

3.3 Evaluation

This section evaluates the write-back policy using YCSB [19], BG [13], and TPC-C [71] benchmarks. All experiments use IQTwemcached [30] that implements S and X leases. With YCSB and BG, we use Design 1. TPC-C uses Design 2. While BG uses MongoDB [55] version 3.4.10 as its data store, both YCSB and TPC-C use MySQL version 5.7.23 as their data store. We chose MongoDB and MySQL to highlight applicability of the write-back policy to both SQL and NoSQL data stores. Moreover, BG highlights the applicability of Design 1 to SQL when the workload is simple interactive social networking actions.

With all benchmarks, we quantify maximum memory used by the write-back policy assuming a sustained high system load. These maximums are unrealistic as a typical workload is diurnal consisting of both a low and a high load, e.g., see Facebook's load [11]. At the same time, it is useful to establish the worst case scenario.

Our experiments were conducted on a cluster of *emulab*[26] nodes. Each node has two 2.4 GHz 64-bit 8-Core (32 virtual cores) E5-2630 Haswell processors, 8.0 GT/s bus speed, 20 MB cache, 64 GB of RAM, connects to the network using 10 Gigabits networking card and runs Ubuntu OS version 16.04 (kernel 4.10.0). Unless stated otherwise, each experiment starts with a warm cache (100% cache hit rate) and runs for 10 minutes.

Obtained results highlight the following lessons:

1. Write-back enhances performance with all benchmarks as long as memory is abundant. It also enhances horizontal scalability as a function of the number of nodes in the caching layer. See Sections 3.3.1, 3.3.2, and 3.3.3.

2. With write-back, there is a tradeoff between the amount of required memory, the number of BGTs applying buffered writes to the data store, and the throughput observed by the application (foreground tasks). Increasing the number of BGTs reduces the amount of memory required by write-back. However, it also reduces the throughput observed by the application. See Sections 3.3.1 and 3.3.2.

3. Limited memory diminishes the performance gains provided by write-back. This is because it must pin buffered writes, mappings, PendingWrites/Queues in memory. These entries may reduce the cache hit rate observed by the application. One approach to mitigate this is to limit the amount of pinned memory allocated to write-back, forcing it to switch to write-through once this memory is exhausted. We observe memcached memory to become calcified [43], preventing write-back from using its assigned memory. See Section 3.3.4.1.

4. The overhead of replicating buffered writes, mappings, PendingWrites/Queues of write-back is in the form of network bandwidth. This overhead becomes negligible when writes are infrequent and the application's cache entries are much larger than these entries. Moreover, an increase in the number of nodes in the caching layer increases the available network bandwidth. This reduces the overall impact of replication. In our experiments, constructing 3 replicas with 4 cache servers reduces observed

throughput with 1 replica by 19%. Experiments conducted with 8 cache servers observes a 6% decrease in throughput. See Section 3.3.4.2.

5. Our proposed write-back technique complements the write-back technique of both a data store such as MongoDB and a host-side cache such as Flashcache. Its enhances their performance more than 2x with workloads that exhibit a high read-write ratio. See Section 3.3.4.3.

Below, we present results in support of these lessons in turn.

3.3.1 YCSB: Design 1 with MySQL

The YCSB database consists of *10 million* records. Each record has 10 fields, each field is 100 bytes in size. A read or an update action reads or updates all fields of a record. A scan action (Workload S) retrieves 5 records (cardinality 5). When all data fits in IQTwemcached, the cache size for Workloads A, B and C is 14 GB. It is 55 GB with Workload S. We drop Workload C from further consideration because it is 100% read and a choice of a write-policy does not impact its performance. Unless stated otherwise, we assume a uniform access pattern. Details of the YCSB workloads are shown in Table 3.3.

Workload	**Actions**
Workload A	50% Read, 50% Update
Workload B	95% Read, 5% Update
Workload C	100% Read
Workload S	95% Scan, 5% Update

Table 3.3: YCSB Workloads.

Response time: Table 3.4 shows response time of YCSB Read, Scan and Update actions with MySQL by itself and alternative write policies. The numbers in parentheses are the number of round-trips from the application server to the data

Action	MySQL	MySQL + Write-Back	MySQL + Write-Through	MySQL + Write-Around
Read	0.20 (1)	0.08 (1)		
Scan	0.36 (1)	0.15 (1)		
Update	0.55 (1)	0.44 (5)	0.92 (4)	0.88 (3)

Table 3.4: Response time (in milliseconds) of YCSB actions.

store and CMIs. The cache expedites processing of Reads and Scans because looking up results is faster than processing a query. Update is faster with write-back because it does not incur the overhead of generating log records. MySQL's response time is with Solid State Disk (SSD) as the mass storage device. With a Hard Disk Drive (HDD), MySQL's response time for Update increases to 35.87 milliseconds.

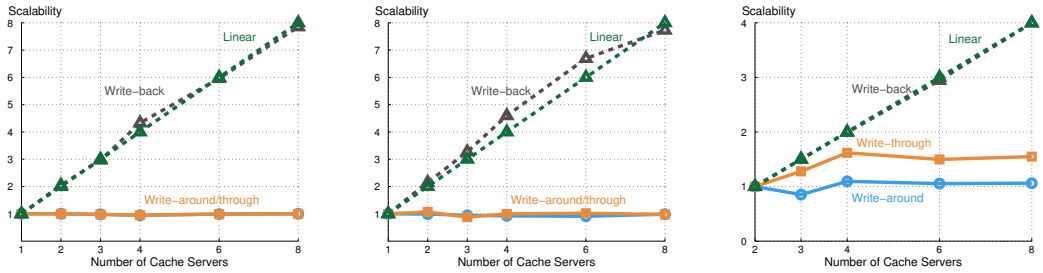
Write-through and write-around are slower than MySQL because they include the overhead of updating the cache. Write-around incurs the following 3 network round-trips: 1) acquire an X lease on the impacted cache entry from the CMI, 2) issue the SQL statement to MySQL, and 3) commit the session. Write-through updates the impacted cache entry using a read-modify-write[2], increasing the number of network round-trips to 4. Write-back increases the number of network round-trips to 5 by writing its buffered write to the CMI[3].

Throughput: Figure 3.1 used as motivation at the beginning of this chapter shows the throughput with the alternative YCSB workloads using write-back, write-around, and write through as a function of cache servers. In these experiments, we prevent the global LRU lock of IQTwemcached from limiting performance by launching 8 instances of cache manager (CMI) per server. Each CMI is

[2]Read of RMW obtains an X lease on the referenced key in addition to reading its value.

[3]Write-back's 5 network round-trips are for as follows. Two round-trips for the RMW of the cache entry. Two round-trips for the RMW of the buffered write. One round-trip for commit that releases the X leases and makes the cache writes permanent.

assigned 8 GB of memory. We increase the number of servers (CMIs) from 1 (8) to 8 (64). Obtained results show that write-back outperforms its alternatives[4] by a wide margin.



(a) Workload A (b) Workload B (c) Workload S

Figure 3.3: Scalability with different YCSB workloads.

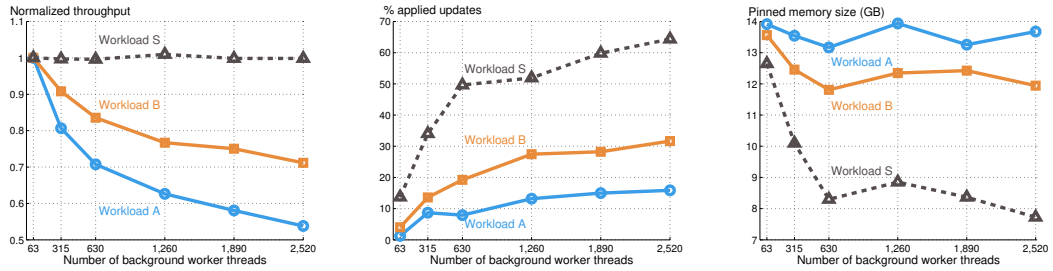
Figure 3.3 shows scalability of Workloads A, B, and S. These correspond to the throughput numbers of Figure 3.1. With Workloads A and B, scalability is computed relative to 1 cache server. Write-around and write-through do not scale because either the data store CPU or disk/SSD becomes the bottleneck. Write-back scales linearly by buffering writes and eliminating the data store from the processing path. While the bottleneck resource is CPU with Workload A, the bottleneck resource is network bandwidth with Workloads B and S.

With Workload S, we show scalability relative to the configuration consisting of 2 cache servers. Its larger cache entries prevent it from observing a 100% cache hit with 1 cache server (and observes a 100% cache hit with 2 or more cache servers). Using 1 cache server as the basis results in a super-linear scaleup. Using 2 servers as the basis of the scalability graph provides for a more subjective evaluation.

[4]Throughput of MySQL by itself is 17,523, 111,156, and 91,499 for Workloads A, B, and S, respectively.

3.3.1.1 Required Memory

The rate at which a system applies buffered writes to the data store is a tradeoff between the cache memory space and decrease in rate of processing (throughput) observed by the foreground requests. The foreground requests are impacted for two reasons. First, background threads compete with foreground threads that observe cache misses for using the data store. These foreground threads must apply their changes to the data store and query it to compute the missing cache entries. Second, application of buffered writes requires network bandwidth for (background threads of) AppNodes to fetch the buffered writes from CMIs. At the same time, an aggressive application of buffered writes deletes both the buffered writes and their mappings from CMIs faster, minimizing the amount of memory required by write-back.



(a) Throughput relative to 63 BGTs (b) % Applied updates (c) Pinned memory size

Figure 3.4: Impact of the number of background workers on throughput, percentage of updates applied to the data store, and the total amount of pinned memory (YCSB workloads).

We demonstrate the tradeoff with an experiment that consists of 63 instances of AppNodes hosted on 21 servers (3 AppNode instance per server), and 64 CMIs hosted on 8 servers (8 CMIs per server). We vary the number of background threads (BGTs) from 1 per AppNode instance to 5, 10, 20, 30, and 40. Each BGT applies buffered writes to the data store as fast as possible.

Figure 3.4a shows the normalized throughput observed with different number of BGTs relative to 1 BGT per AppNode instance (63 BGTs). The x-axis of this figure is the total number of BGTs. Increasing the number of BGTs decreases throughput of write-heavy workload A (47% drop with 2,520 BGTs) followed by read-heavy Workload B (30% drop with 2,520 BGTs). Workload S observes a negligible decrease in its throughput even though it is a read-heavy workload similar to B. Moreover, the size of buffered writes with both B and S are identical. S is different because it consists of scans with cache entries that are 5 times larger than those of B. The network bandwidth becomes the bottleneck with S to render the overhead of additional BGTs negligible.

Figure 3.4b shows the percentage of changes applied by the background threads to the data store at the end of an experiment. Workload A has the lowest percentage because it is write-heavy. However, this percentage increases modestly ($< 20\%$) as we increase the number of BGTs. This explains why Workload A has the highest amount of pinned memory in Figure 3.4c.

Figure 3.4 shows Workload S benefits the most from an increase in the number of BGTs. This is because its normalized throughput is comparable to having 63 BGTs while its percentage of applied writes to the data store increases dramatically. This in turn reduces its pinned memory size by almost 2x.

3.3.2 BG: Design 1 with MongoDB

BG [13] is a benchmark that emulates interactive social networking actions. It quantifies Social Action Rating (SoAR) defined as the highest throughput that satisfies a pre-specified service level agreement (SLA). The SLA used in our experiments is 95% of actions processed in 100 milliseconds or faster. In our evaluation, we use a social graph consisting of 10 million users with 100 friends per user.

We considered three workloads as shown in Table 3.5. List friends (or list pending friends) action only returns 10 out of 100 friends (or pending friends) of the requested user.

BG Social Actions	**90% reads**	**99% reads**	**99.9% reads**
View Profile	80%	89%	89.9%
List Friends	5%	5%	5%
List Pending Friends	5%	5%	5%
Invite Friend	4%	0.4%	0.04%
Reject Friend	2%	0.2%	0.02%
Accept Friend	2%	0.2%	0.02%
Thaw Friendship	2%	0.2%	0.02%

Table 3.5: BG Workloads.

Response time: Table 3.6 shows the response time of BG's actions with MongoDB by itself and with IQTwemcached using write-back, write through, and write-around. MongoDB was configured to acknowledge writes immediately after writing it in its memory, flushing dirty documents to its mass storage device very 60 seconds (writeConcern=ACKNOWLEDGED). These numbers were gathered using a single threaded BG generating requests.

While the cache improves the response time of the read actions with alternative write policies, its response time for the write actions is worse than MongoDB by itself. This is due to a higher number of round-trips to the cache, see the numbers in parentheses. If MongoDB was configured to generate log records and flush them to disk prior to acknowledging the write (writeConcern=JOURNALED), write-back would have been faster [5]

[5]MongoDB with writeConcern=JOURNALED provides the following response time: 0.6 milliseconds for each of Invite Friend and Reject Friend, and 1.2 milliseconds for each of Accept Friend and Thaw Friendship.

Write-back performs the write actions faster than write-through and write-around because it performs the write to MongoDB asynchronously. Write-around outperforms write-through by reducing the number of round-trip messages because deleting impacted cache entries requires 1 round-trip versus 2 round-trips to update a cache entry using RMW with write-through.

One may use Table 3.6 in combination with the frequencies shown in Table 3.5 to compute the average response time. With BG's lowest read-write ratio of 90%, the average response time with write-back (0.26 msec) is faster than MongoDB by itself (0.53 msec), with write-through (0.30 msec) and write-around (0.29 msec). Alternative write policies are faster than MongoDB because the View Profile action dominates (89%) the workload and the cache improves its performance dramatically. With BG's high read-write ratio of 99.9%, write-back becomes significantly faster than MongoDB by itself. Moreover, write-through and write-around provide a response time comparable to write-back.

Action	MongoDB	MongoDB + Write-Back	MongoDB + Write-Through	MongoDB + Write-Around
View Profile	0.52 (1)	0.18 (1)		
List Friends / Pending Friends	0.81 (1)	0.52 (2)		
Invite Friend	0.23 (1)	0.34 (5)	0.70 (5)	0.62 (4)
Reject Friend	0.24 (1)	0.41 (6)	0.71 (6)	0.66 (4)
Accept Friend	0.46 (2)	1.01 (11)	1.64 (11)	1.41 (6)
Thaw Friendship	0.48 (2)	0.87 (11)	1.66 (11)	1.21 (5)

Table 3.6: Response time (in milliseconds) of BG actions with MongoDB in acknowledged mode, writeConcern=ACKNOWLEDGED. Number of network round-trips is shown in parentheses.

Throughput: Write-back provides a higher SoAR when compared with other policies, see Table 3.7. Its SoAR is dictated by the network bandwidth of the cache server. Moreover, it enables all BG workloads to scale linearly as a function of

cache servers. Write-around and write-through scale sub-linearly by no more than a factor of 6 with 8 cache servers. Scalability is lowest with 90% reads because the CPU of MongoDB becomes fully utilized processing writes. Scalability improves with 99% and 99.9% reads as the load on MongoDB is reduced and the network card of the cache servers becomes fully utilized.

Workload	**SoAR (actions/sec)**		
	MongoDB	Write-back	Write-through
90% read	27,518	355,769	111,548
99% read	48,365	693,816	488,593
99.9% read	76,068	711,659	678,665

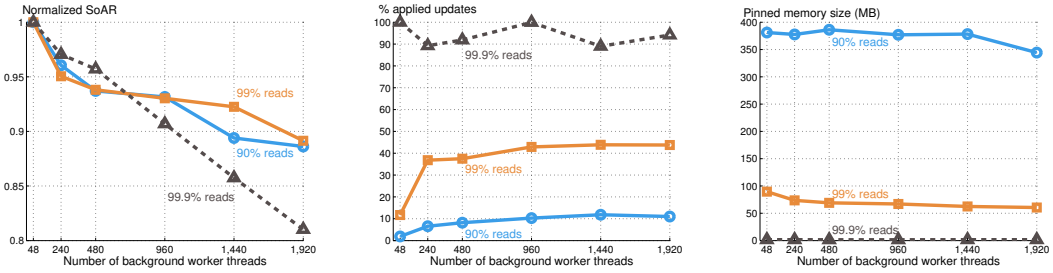
Table 3.7: SoAR of MongoDB by itself and with 1 cache server.

3.3.2.1 Required Memory

Similar to YCSB results, the amount of memory required for buffered writes with BG is a function of their production rate by the foreground threads and application to the data store by BGTs. Obtained results are shown in Figure 3.5. A key difference is that with its read-heavy (99.9%) workload, the BGTs apply buffered writes to the data store at the same rate at which writes produce them. The measured SOAR is impacted by the overhead of BGTs checking for buffered writes to find none. This is because the network is the bottleneck. Hence, increasing the number of BGTs reduces throughput without providing a benefit, see Figure 3.5a.

3.3.3 TPC-C: Design 2 with MySQL

TPC Benchmark C [71] is an on-line transaction processing (OLTP) benchmark. TPC-C models a supplier operating out of several warehouses and their associated sales districts. It quantifies transactions per minute (tpmC) rating of a solution. We use the standard setting of TPC-C with 1 warehouse consisting of



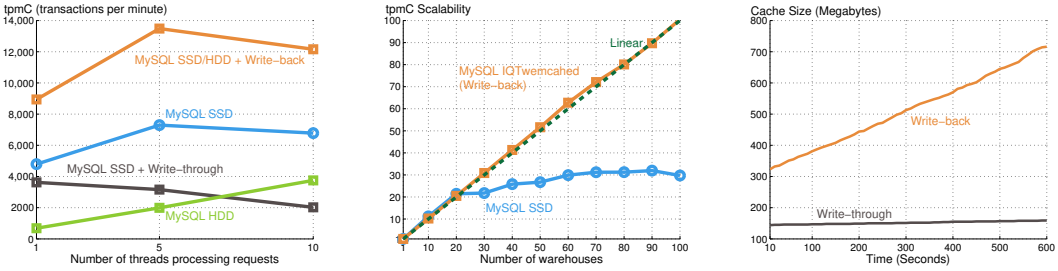
(a) SoAR relative to 48 BGTs

(b) % Applied updates

(c) Pinned memory size

Figure 3.5: Impact of the number of background workers on SoAR, percentage of updates applied to the data store, and the total amount of pinned memory (BG workloads).

ten districts per warehouse, each district serving three thousand customers, and 100,000 items per warehouse. These results are obtained using OLTP-Bench [23] implementation of TPC-C extended with leases for strong consistency. The deployment consists of 2 emulab nodes. One hosting the OLTP-Bench workload generator and the other hosting (a) MySQL by itself and (b) MySQL and 8 instances of IQTwemcached using either write-through or write-back policies.



(a) tpmC of MySQL by itself and with IQTwemcached configured with write-back and write-through.

(b) tpmC Scalability.

(c) Memory size of IQTwemcached with write-back and write-through, 5 TPC-C threads, 1 warehouse.

Figure 3.6: tpmC of MySQL by itself and with IQTwemcached configured with write-back and write-through.

We analyze MySQL configured with either solid state drive (SSD) or hard disk drive (HDD). Write-back improves performance throughput of MySQL with

Transaction	MySQL	MySQL+Write-Back	MySQL+Write-Through
New-Order	8.06	5.18	10.99
Payment	2.09	1.27	3.29
Order-Status	1.59	0.71	
Delivery	22.36	3.56	24.09
Stock-Level	2.00	1.35	

Table 3.8: Response time (in milliseconds) of TPCC actions.

SSD by more than two folds. Moreover, its performance is not sensitive to whether MySQL is configured with either SSD or HDD because TPC-C's database is small, causing cache to observe a 100% hit rate [6].

Response time: Table 3.8 shows the average response time of each TPC-C transaction using MySQL by itself, with write-back, and write-through. Write-back improves the response time of all transactions. Write-through improves the response time of read-only transactions: Order-Status and Stock-Level. However, write-through causes the write transactions to incur the overhead of updating both the cache entries and MySQL. Hence, it is slower than MySQL by itself and with write-back.

The frequency of New-Order, Payment, Order-Status, Delivery, and Stock-Level transactions are 45%, 43%, 4%, 4%, and 4%. The weighted response time with MySQL is 5.56 milliseconds. Write-back is faster at 3.10 milliseconds, providing a 44% enhancement. Write-through is 33% slower than MySQL because the overhead of writes outweighs the benefits observed by the reads that constitute 8% of the workload.

Throughput: Figure 3.6a shows the write-through policy is inferior to MySQL by itself because 92% of TPC-C transactions are writes. Write-through must apply

[6]MySQL with HDD slows down the rate at which BGTs apply buffered writes, requiring more memory than MySQL with SSD.

these writes to both MySQL and IQ-Twemcached synchronously. The overhead of writing to IQ-Twemcached outweighs benefits provided by its cache hits.

In Figure 3.6a, tpmC of write-back levels off with 5 and more threads due to contention for leases between the foreground threads. These foreground threads also contend with the background thread for an X lease on the Queue of sessions. The same applies to MySQL with its lock manager blocking transactions to wait for one another.

Figure 3.6b shows scalability of TPC-C with MySQL and write-back as we increase the number of warehouses from 1 to 100 with 1 thread issuing transactions to one warehouse. MySQL's SSD becomes fully utilized with more than 20 threads (warehouses), limiting its scalability. Write-back scales linearly with network bandwidth of the caching tier dictating its scalability.

3.3.3.1 Required Memory

Figure 3.6c shows the amount of required memory with write-through and write-back policies with 1 warehouse. These results highlights the amount of memory required by the cache entries for TPC-C. Its increase with write-through as a function of time highlights the growing size of database (new orders) and its corresponding cache entries. These entries are included in the memory size reported for write-back. Moreover, write-back includes buffered writes, mappings, and Queues. The difference between write-back and write-through highlights (a) the extra memory required by write-back for buffered writes and (b) faster processing of new orders increases both database size and cache size at a faster rate. Write-back requires a significantly higher amount of memory than write-through.

3.3.4 Discussion

This section discusses impact of limited memory on write-back, overhead of replicating buffered writes for durability, and tradeoffs associated with deploying write-back at different software layers.

3.3.4.1 Limited Memory and Slab Calcification

Reported performance numbers assume abundant amount of memory. Write-back performance degrades considerably with limited memory because (a) buffered writes compete with cache entries for memory to increase the cache miss rate observed by the application, (b) cache misses require buffered writes to be applied to the data store in a synchronous manner that diminishes the performance of write-back to be comparable to write-though[7], (c) cache managers such as memcached may suffer from slab calcification when memory is limited. We highlight these using YCSB.

This section illustrates the impact of limited memory with Design 1 using YCSB workload. We assume 16 AppNodes each with 10 BGTs, 1 MongoDB[8] server, 1 cache server with 8 CMI instances and a total of 14 GB of memory. This 14 GB cache space is enough for YCSB Workloads A and B to observe a 100% cache hit rate with the write-through policy. With the write-back policy, both the buffered writes and mappings start to compete with the cache entries for space. We vary the maximum amount of memory that can be occupied by these pinned key-value pairs to quantify the impact of limited memory on write-back when compared with write-through.

[7]Figure 3.6a shows write-through to be inferior to MySQL by itself with TPC-C.

[8]Our switch from MySQL of Section 3.3.1 with YCSB to MongoDB is intentional to highlight flexibility of Design 1 to work with alternative data models.

Figure 3.7 shows the throughput of Workload A increases as we increase the size of pinned memory from 2 to 12 GB. Logically, an increase in the value of x-axis leaves a lower amount of memory for regular key-value pairs to observe cache hits. Consider results shown for Workloads A and B in turn.

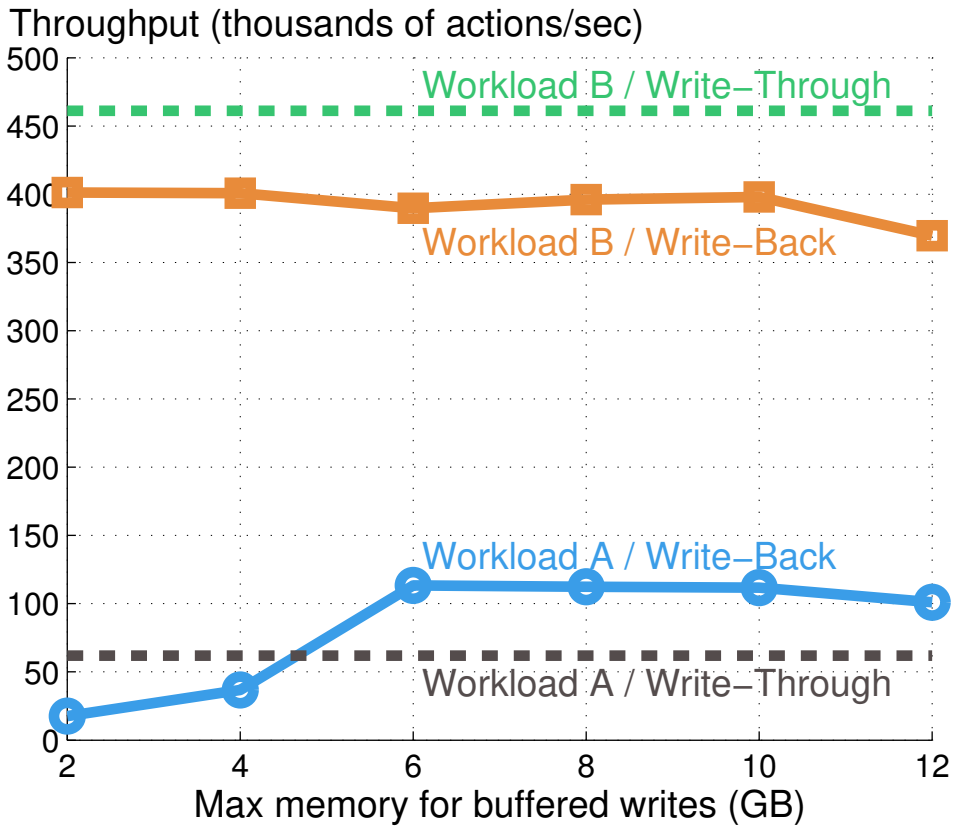


Figure 3.7: Impact of limited memory on write-back performance. Cache memory size is 14 GB. YCSB Zipfian constant is 0.75.

The throughput of Workload A is 52K with write-through and independent of the x-axis values. With 2 GB of pinned memory, write-back exhausts this memory with buffered writes quickly. Every subsequent write generates its buffered write for the cache only to be rejected, causing the write to update the data store synchronously (switches to write-through). Repeated failed attempts to insert buffered writes lowers write-back throughput, enabling write-through to outperform it. As

we increase the size of pinned memory, a larger fraction of writes are performed using write-back, enabling write-back to outperform write-through. However, this reduces cache hit rate from 87% with 2 GB to 48% with 12 GB of pinned memory due to smaller memory size for regular key-value pairs that service read requests. Hence, the throughput observed with write-back levels-off, outperforming write-through by 2x. (If the memory was increased to 48 GB then write-back throughput would have been 6x higher.)

With 12 GB of pinned memory, write-back pins only 10 GB of data for Workload A. This is due to slab calcification [43, 14, 18]. To elaborate, memcached constructs two slab classes: Class 1 stores mappings. Class 2 stores regular key-value pairs and the buffered writes as they are approximately the same size[9]. Once memory is assigned in its entirety, these two slab classes are calcified and may not allocate memory from one another. Once memory space assigned to Class 1 is exhausted, no additional mappings may be inserted in the cache. This means a subsequent write must be processed synchronously even though there is space[10] in Class 2 for buffered writes.

The slab calcification phenomena is magnified with Workload B, causing write-back to provide a lower performance than write-through for the entire spectrum of maximum pinned memory sizes. Only 5% of Workload B is writes, causing the slab class for pinned mappings to be significantly smaller. Thus, only 5.6 GB of memory can be used for buffered writes even though maximum pinned memory size is significantly larger, i.e., x-axis values of 6, 8, 10 and 12 GB.

The degree of skew in access pattern to data items is beneficial to write-back and impacts the observed trends. It increases the likelihood of several writes impacting

[9]A write updates all properties of a document.

[10]This space does not sit idle and is occuped by the application's cache entries.

the same MongoDB document. The write-back policy merges their changes into one pinned buffered write. Moreover, the number of mappings is reduced. Both minimize the number of pinned key-value pairs. In turn this results in a higher cache hit[11] rate, enhancing throughput by 40-50% with both workloads and a very skewed access pattern. Now, write-back outperforms write-through with both workloads.

A future research direction is to compare write-back with write-through using cache managers that do not suffer from slab calcification. Another research direction is to extend write-back to a dynamic framework that detects limited memory and switches to write-through, preventing a degradation in system performance. Alternatively, an algorithm may adjust the size of pinned memory (buffered writes and mappings) to maximize throughput.

3.3.4.2 Replication of Buffered Writes

Replication of buffered writes enhances their availability in the presence of cache server failures. At the same time, it consumes network bandwidth to transmit these redundant replicas to CMIs and the additional memory required to store them. While only one replica is fetched by a BGT to apply to the data store, the BGT must delete all replicas. This overhead impacts system throughput. This section quantifies this overhead by comparing the throughput of the write-back configured with 1 and 3 replicas for buffered writes, their mappings, and PendingWrites/Queues.

Obtained results highlight the following lesson. The overhead of constructing 3 replicas becomes less significant as we 1) increase the size of the caching layer,

[11]Workload B observes a 10% increase in cache hit rate.

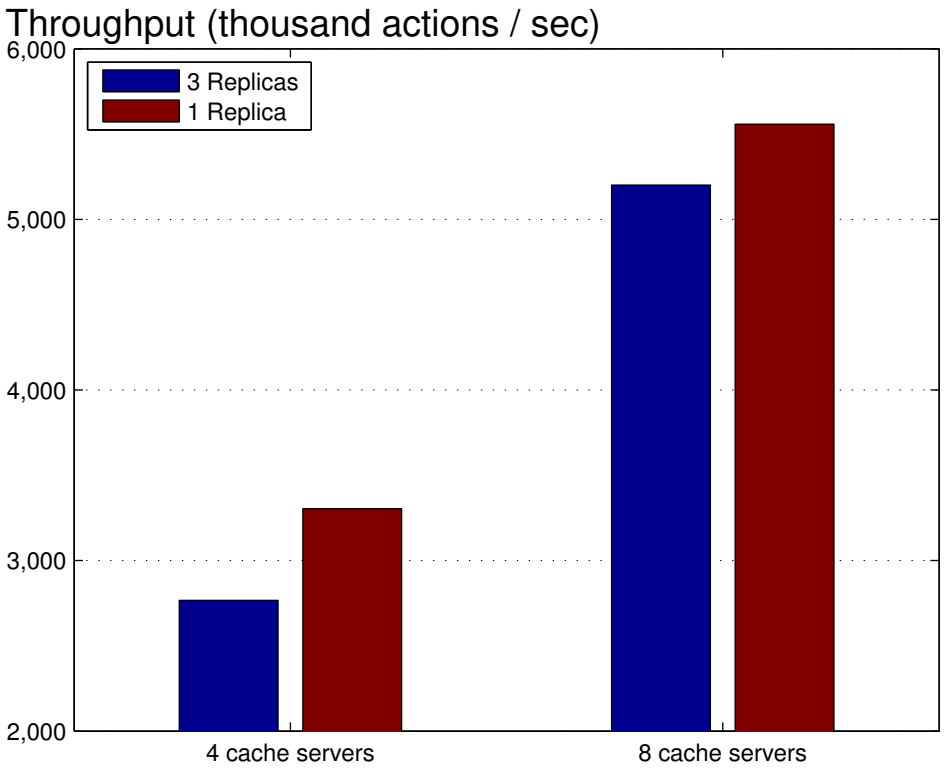


Figure 3.8: Impact of replicating buffered writes on throughput of YCSB Workload B.

2) reduce the frequency of writes and 3) have workloads with cache entries much larger than the buffered writes and their mappings.

Figure 3.8 highlights the above lessons by showing the throughput of YCSB workload B with 4 and 8 cache servers using Design 1. (Each cache server hosts 8 CMIs.) The overhead of constructing 3 replicas with 4 cache servers lowers throughput by 19%. It is reduced to 6% with 8 cache servers. The larger configuration has a higher network bandwidth, reducing the overhead of replication more than three folds.

With Workload S, the impact of replicating buffered writes is not noticeable with both configurations. This is because writes are 5% of the workload and the

size of buffered writes and their mappings is insignificant relative to cache entry sizes. The network bandwidth limits the throughput of this workload with both 1 and 3 replicas.

3.3.4.3 Comparison with Alternative Write-Back Caches

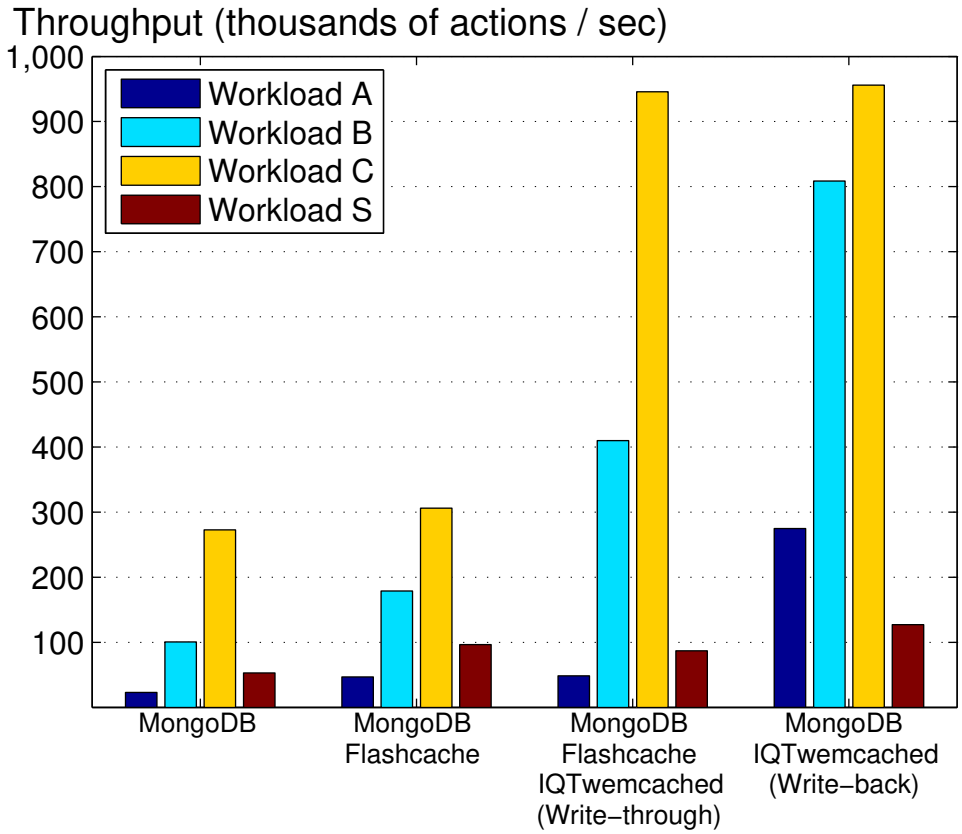


Figure 3.9: Performance of alternative cache configurations with YCSB. MongoDB is configured with writeConcern set to ACKNOWLEDGED.

With CADS architecture, one may apply the write-back policy in different software layers. For example, MongoDB implements write-back by acknowledging a write as soon as it is stored in its buffers (writeConcern is set to ACKNOWLEDGED). It flushes buffered writes to disk every 60 seconds. These buffered writes are not durable and if MongoDB crashes then they are lost.

Host-side caches stage disk pages referenced by a data store such as MongoDB in SSD to enhance performance. They are typically deployed seamlessly using a storage stack middleware or the operating system [20, 22, 67, 52, 17, 68, 39, 47, 44]. Examples include Flashcache [52] and bcache [67]. One may configure a host-side cache with alternative write-policies.

While these caches complement the application-side cache [4] and its write-back policy, a key question is how do they compare with one another? This section shows write-back using the application-side cache outperforms the other alternative by several orders of magnitude with both YCSB and BG. This is true even when the other two types of caches are combined together. Below, we present results using YCSB. BG provides similar observations.

Figure 3.9 compares the performance of four cache configurations using four YCSB workloads. The alternative configurations include: MongoDB, MongoDB with Flashcache, MongoDB with Flashcache and IQTwemcached using write-through, and MongoDB with IQTwemcached using write-back. In all configurations, MongoDB is configured with writeConcern set to ACKNOWLEDGED. The four YCSB workloads are shown in Table 3.3. Results are obtained using 1 server for MongoDB and its Flashcache, 1 server for IQTwemcached, and 8 AppNode servers generating requests. Consider results of each workload in turn.

YCSB Workload C is 100% read and Figure 3.9 highlights the benefits of using an application-side cache when compared with MongoDB either by itself or Flashcache. Application-side cache enhances throughput more than 3 folds regardless of the write-policy. The cache provides for result look up instead of MongoDB processing a query, improving throughput dramatically.

Workload B benefits from the application-side cache because 95% of its requests are identical to Workload C. The remaining 5% are writes that benefit from the write-back policy, enabling it to out-perform write-through almost 2x.

YCSB Workload A is write-heavy with 50% update. MongoDB performance with Flashcache is enhanced 2 folds because writes with SSD are faster than HDD. Using Linux fio benchmark, we observe the SSD IOPS for 4K block size to be 1.14x higher than HDD for sequential reads, 1.17x for sequential writes, 150x for random reads, and 50x for random writes. Every time MongoDB flushes pending writes to disk using fsync, it blocks write operations until fsync completes. Using SSD instead of HDD expedites fsync to improve performance.

Workload A does not benefit from an application-side cache configured with the write-through policy. However, its throughput is improved more than 5x with the write-back policy because writes are buffered in IQTwemcached and applied to MongoDB asynchronously.

Workload S utilizes network bandwidth of the cache server fully with both write-through and write-back policies. The improvement it observes from using write-back is 30% because 5% of its requests are writes. It is interesting to note that MongoDB with Flashcache provides a comparable throughput to the write-through policy, rendering the application-side ineffective for this workload. The explanation for this is that the network bandwidth of MongoDB with Flashcache becomes fully utilized with Workload S, enabling it to provide a comparable throughput.

3.4 Proof of Read-after-write Consistency

Below, we provide a formal proof of read-after-write consistency for the write-back policy. We assume (a) a developer implements a write session correctly,

transitioning the data store from one consistent state to another and (b) the data store provides strong consistency.

Definition 3.4.1. *A read R_i may be served by a cache entry represented as a key-value pair (k_i, v_i) where k_i identifies the entry and v_i is the value of the entry. Both k_i and v_i are application specific and authored by a developer.*

Definition 3.4.2. *A write W_j that impacts the value v_i of key k_i updates v_i to reflect the changes of W_j.*

Definition 3.4.3. *A buffered write for a document D_i is represented as a key-value pair (k_i^{bw}, v_i^{bw}) where k_i^{bw} identifies a buffered write and v_i^{bw} stores either the final value of D_i or the pending changes to D_i. With the latter, the sequence of changes in v_i^{bw} represents the serial order of writes to D_i. This serial order is dictated either by the arrival time of writes or Quarantine leases of the cache manager.*

Definition 3.4.4. *A* mapping *inputs the key k_i for a cache entry to compute the keys $\{k_i^{bw}\}$ of the buffered writes. Without loss of generality, we assume the function f is $f(k_i) \rightarrow \{k_i^{bw}\}$.*

Theorem 1. *The write-back policy provides read-after-write consistency while applying pending buffered writes to the data store.*

We break down the theorem into the following three lemmas:

Lemma 1.1. *A read session R_i observes the values produced by the last write session that committed prior to its start.*

Proof. If v_i exists, based on Definition 3.4.2, those writes impacting k_i that commit prior to the start of R_i updated v_i to reflect their value. R_i observes a cache hit and consumes v_i, reflecting the value produced by the latest write session. Should

R_i observe a cache miss, it looks up $\{k_i^{bw}\}$ from the mapping using k_i. For each k_i^{bw}, it applies v_i^{bw} to the data store. R_i queries the data store to compute the missing v_i and puts the resulting k_i-v_i in the cache. v_i includes changes of all write sessions that commit prior to R_i. While R_i applies v_i^{bw}, all concurrent writes that impact v_i try to acquire a Q lease on k_i^{bw}, abort, and try again until R_i releases its Q lease on k_i^{bw}. These writes are serialized after R_i. ■

Lemma 1.2. *Concurrent read and write sessions that reference the same buffered write are serialized.*

Proof. R_i and W_j are two concurrent sessions that impact the same key k_i. Two serial schedules are possible: R_i either before or after W_j. Both R_i and W_j must acquire a Q lease on the same buffered write. Q leases are not compatible and only one is granted the lease. R_i races with W_j for the Q lease and the winner is serialized before the other.

■

Lemma 1.3. *Concurrent write sessions that reference the same buffered write are serialized.*

Proof. Concurrent write sessions that impact the same buffered write k_i^{bw} must acquire Q leases on k_i^{bw}. Only one may proceed at a time because Q leases are incompatible with each other on the same key, dictating their order. When one acquires a Q lease successfully, it incorporates its changes to v_i^{bw} and releases the Q lease. ■

Theorem 2. *The write-back policy maintains consistency of the data store in the presence of arbitrary failures of threads that apply buffered writes to the data store.*

Proof. A thread that applies a buffered write to the data store may be a background thread or a read R_i that observes a cache miss. These threads acquire a Q lease on

the buffered write and apply its changes to the data store. Failures of these threads cause their Q lease to time out, making the buffered write available again. The buffered write may either be idempotent or non-idempotent. If it is idempotent, its repeated application to the data store does not compromise data store consistency. If it is non-idempotent, it has not yet been applied to the data store. A buffered write is always replaced with its idempotent equivalent prior to being applied to the data store. Hence, consistency of the data store is preserved. ■

Chapter 4

TARDIS

A read or a write request to the data store *fails* when it is not processed in a timely manner. This may result in denial of service for the end user. All requests issued to the data store fail when it is unavailable due to either hardware or software failures, natural disasters, power outages, human errors, and others. It is possible for a subset of data store requests to fail when the data store is either sharded or offered as a service. With a sharded data store, the failed requests may reference shards that are either unavailable or slow due to load imbalance and background tasks such as backup. With data store as a service, requests fail when the application exhausts its pre-allocated capacity. For example, with the Amazon DynamoDB and Google Cloud Datastore, the application must specify its read and write request rate (i.e., capacity) in advance with writes costing more [6, 33]. If the application exhausts its pre-specified write capacity then its writes fail while its reads succeed.

An application may process its failed writes in a variety of ways. Simplest is to report the failures to the end users and system administrators. The obvious drawback of this approach is that the write fails and the end user is made aware of this to try again (or for the system administrator to perform some corrective action). Another variant is for the application to ignore the failed write. To illustrate, consider a failed write pertaining to Member A accepting Member B's friend invitation. The system may simply ignore this write operation and continue to show B's invitation to A to be accepted again. Assuming the failure of the data

Failure Duration	Read to Write Ratio	
	100:1	1000:1
1 min	5,957	561
5 min	34,019	3,070
10 min	71,359	6,383

Table 4.1: Number of failed writes with different BG workloads and data store failure durations. TARDIS using CAMP persists all failed writes.

store is short-lived then the user's re-try may succeed. This alternative loses those writes that are not recoverable. For example, when Member C invites Member B to be friends, dropping this write silently may not be displayed in an intuitive way for the end user to try again. A more complex approach is for the application to buffer the failed writes. A system administrator would subsequently process and recover these writes.

We seek for an automated solution that replaces the administrator with a smart algorithm that buffers failed writes and applies them to the data store at a later time once it is available. This approach constitutes the focus of TARDIS[1], a family of techniques for processing failed writes that are transparent to the user issuing actions.

TARDIS teleports failed writes by buffering them in the cache and performing them once the data store is available. Its general form employs off-the-shelf cache managers with some probability of losing acknowledge writes due to either evictions (insufficient memory) or failure of the cache. The cache manager may be tailored to minimize this probability to zero.

Table 4.1 shows the number of failed writes with different failure durations using a benchmark for interactive social networking actions named BG [13]. We

[1]Time and Relative Dimension in Space, TARDIS, is a fictional time machine and spacecraft that appears in the British science fiction television show Doctor Who.

consider two workloads with different read to write ratios. TARDIS teleports all failed writes and persists them once the data store becomes available.

TARDIS addresses the following challenges:

1. How to process data store reads with pending buffered writes in the cache?

2. How to apply buffered writes to the data store while servicing end user requests in a timely manner?

3. How to process non-idempotent buffered writes with repeated failures during recovery phase?

4. What techniques to employ to enable TARDIS to scale? TARDIS must distribute load of failed writes evenly across the cache instances. Moreover, its imposed overhead must be minimal and independent of the number of cache servers.

5. How to provide read-after-write consistency with different views of AppNodes. One AppNode may see the data store available while another does not.

TARDIS preserves consistency guarantees of its target application while teleporting failed writes. This is a significant improvement when compared with today's state of the art that loses failed writes always.

Advantages of TARDIS are two folds. First, it buffers failed writes in the cache when the data store is unavailable and applies them to the data store once it becomes available. Second, it enhances productivity of application developers and system administrators by providing a universal framework to process failed writes. This saves both time and money by minimizing complexity of the application software.

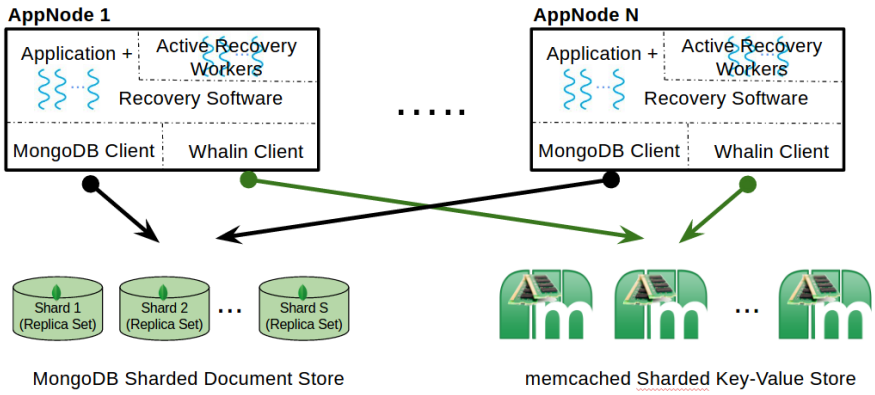


Figure 4.1: TARDIS architecture.

To simplify the discussion, the assumptions of data model for our design of TARDIS is identical with the assumptions of Deign 1 of the write-back policy (see Chapter 3). Assumptions of TARDIS include:

- AppNodes and the cache servers are in the same data center, communicating using a low latency network. This is a reasonable assumption because caches are deployed to enhance AppNode performance.
- The cache is available to an AppNode when data store writes fail.
- A developer authors software to reconstruct a data store document using one or more cached key-value pairs. This is the recovery software shown in Figure 4.1 used by both the application software and Active Recovery (AR) workers.
- The data store write operations are at the granularity of a single document and transition its state from one consistent state to another. In essence, the correctness of data store writes are the responsibility of the application developer.

- There is no dependence between two or more buffered writes applied to *different*[2] documents. This is consistent with the design of a document store such as MongoDB to scale horizontally.

Contributions of this work are as follows. First, we present the design of TARDIS, a general purpose framework for CADS architecture. This includes the following novel design decisions. An AppNode maintains no buffered writes or database state, allowing a request to be directed to any AppNode for processing. All state information is maintained using key-value pairs stored in the cache and visible to all AppNodes. Moreover, multiple AppNode servers are not required to have a consistent view of the data store and whether it is available or unavailable: One AppNode may operate assuming the data store is unavailable while another may operate assuming it is available. This eliminates the need for synchronization among AppNodes that implement TARDIS, enabling the system to scale.

Second, TARDIS does not compromise consistency guarantees of its target application as long as its buffered writes are not lost - this is a significant improvement over today's state of the art that loses failed writes always.

Third, TARDIS does not prevent all undesirable race conditions because they would slow the system down during normal and recovery modes. Instead, it detects their occurrence and resolves them, e.g., see discussion of Step 6 of Algorithm 6.

Fourth, we present two alternative implementations of TARDIS named Recon and Contextual. Recon requires no modifications to memcached and its design works with an off-the-shelf cache manager that implements interfaces similar to memcached. Contextual requires modifications to the cache manager. Both exhibit similar performance characteristics.

[2]TARDIS preserves the order of two or more writes for the same document.

4.1 Overview

TARDIS consists of four techniques named TAR, DIS, TARD, and TARDIS. The last technique embodies the capabilities of the first three. Hence, the entire family is named TARDIS.

When the data store is unavailable, TARDIS techniques use the caching layer to stage buffered writes. Once the data store becomes available, these techniques apply the buffered writes to the data store. Their primary advantage is enhanced availability of the system for processing writes when the data store is unavailable, using the cache to teleport the write by performing it in the future instead of reporting a failure. This reduces the overall system downtime. A disadvantage of these techniques is their increased application software complexity to ensure read-after-write consistency [50] when the data store recovers since a data item may have buffered writes in the cache and its value in the data store is stale.

TARDIS family of techniques operate in three distinct modes: failed, recovery, and normal. Availability of the data store dictates the mode of operation. The data store is unavailable in failed mode and available in both recovery and normal modes, see Table 4.2 and Table 4.3. Alternative techniques differ in how they process application reads and writes in failed and recovery modes. This dictates the complexity of technique. TAR is the least complex. Complexity increases with DIS, TARD, and TARDIS. All techniques assume background Active Recovery (AR) threads apply buffered writes to the data store once it becomes available after a failure. Below, we present alternative techniques in turn.

In recovery mode, TAR requires application writes to continue to buffer their writes while background AR threads apply them to the data store. With TAR, recovery mode ends once AR workers apply all buffered writes to the data store.

Table 4.2: Processing of cache misses and writes in failed mode.

	Cache miss	Write
TAR/DIS	Unavailable.	Generate buffered writes.
TARD/TARDIS	Unavailable.	Generate buffered writes and a mapping between impacted cache entries and the buffered writes.

Table 4.3: Processing of cache misses and writes in recovery mode.

	Cache miss	Write
TAR	Unavailable.	Generate buffered writes.
TARD	Apply buffered writes and read the updated value from the data store.	Generate buffered writes.
DIS	Unavailable.	Apply buffered writes and update the data store.
TARDIS	Apply buffered writes and read the updated value from the data store.	Apply buffered writes and update the data store.

While TAR is simple to implement, it may result in a long recovery duration. Moreover, in recovery mode, TAR may not process cache misses using the data store. Otherwise, it may observe read-after-write consistency violations for those cache entries with buffered writes that have not yet been applied to the data store. This may be acceptable for certain class of applications. For those that require read-after-write consistency, possible solutions include suspending processing of writes until all buffered writes have been applied to the data store, or switching to either TARD or TARDIS that requires additional software to process these cache misses in a consistent manner.

DIS extends TAR by requiring a write to apply relevant buffered writes to the data store prior to performing its write. A relevant buffered write is one that impacts the same data item(s) in the data store. The application write may merge the buffered writes with its own into one data store write operation, producing

the final value of a data item. DIS is an improvement on TAR because it prevents application writes from generating buffered writes in recovery mode. Instead, these writes apply buffered writes to the data store, ending recovery mode sooner than TAR. We report on several experiments where TAR does not complete recovery and DIS completes recovery in few seconds to tens of seconds depending on the foreground load. DIS is similar to TAR in that it does not process cache misses in recovery mode to provide strong consistency, reporting an error. By completing recovery faster than TAR, DIS switches the system to normal mode. Hence, DIS enables the system to process cache misses sooner than TAR relative to when the data store recovers.

TARD extends TAR by requiring the developer to provide a *mapping* from a cache entry to the buffered writes. In recovery mode, a read that observes a cache miss uses this mapping to identify buffered writes impacting the referenced cache entry. It applies these buffered writes to the data store, queries the data store for the missing key-value pair, computes the cache entry, and inserts it in the cache for future reference. TARD is different from DIS in that it requires reads that observe a cache miss to apply buffered writes to the data store. DIS is different from TARD by requiring writes to apply buffered writes to the data store.

TARD is more complex than DIS because it requires the application software to provide the mapping and manage it. DIS does not have this requirement because a write action has the identity of buffered writes readily available (since it produces buffered writes in failed mode). If cache misses are very rare or non-existent then the recovery duration becomes very long with TARD. This causes writes to generate buffered writes even though the data store is available. Surprisingly, experimental results show this improves the application performance as generating buffered writes are significantly faster than performing the actual write using the

Table 4.4: List of terms and their definition.

Term	Definition
AR	Active Recovery worker migrates buffered writes to the data store eagerly.
D_i	A document in the data store identified by a primary key P_i.
P_i	Primary key of document D_i. Also referred to as document id.
$\{K_j\}$	A set of key-value pairs associated with a document D_i.
Δ_i	A key whose value is a set of changes to document D_i.
TeleW	A key whose value contains P_i of documents with teleported writes.
ω	Number of TeleW keys.

data store, see discussions of Figure 4.8. This interesting observation holds true as long as there is abundant memory because the buffered writes are pinned in memory to prevent their evictions.

TARDIS combines DIS and TARD into one technique. Identical to DIS, it requires writes to apply buffered writes to the data store prior to performing their write. Identical to TARD, it requires the developer to provide a mapping that is used by reads that observe a cache miss to apply relevant buffered writes (if any) to the data store prior to computing the cache entry. It finishes recovery faster than the other algorithms because foreground requests are active participants in applying the buffered writes to the data store. It is also complex to implement because it requires the developer to provide software for reads and writes to apply buffered writes to the data store. In failed mode, TARDIS can be compared to a write-back policy (see Chapter 3) because it buffers writes and postpones them to be applied at some point in the future. In recovery mode, TARDIS cannot be compared to a write-back policy because it performs writes synchronously.

4.2 Writes and Failures

The CADS architecture may process a write using either a write-around, write-through, or a write-back policy. Write-around deletes the cache entry impacted by the write. Both write-through and write-back update the impacted cache entry. Another difference between these techniques is how they perform the write using the data store. Write-around and write-through update the data store synchronously. Write-back updates the data store asynchronously. The techniques described in this chapter apply to those write policies that perform the write synchronously, namely, write-around and write-through policies. They do not apply to write-back because its asynchronous application of writes tolerates data store failures. As detailed in Chapter 3, implementing a write-back policy with CADS is complex. This is especially true when the application requires durability of writes. For those application writes that cannot be implemented using write-back (due to excessive complexity), one may use a simple TARDIS technique to perform the write even though the data store is available.

Definition 4.2.1. *A failed write is a write request to the data store that is not processed in a timely manner, resulting in an exception.*

A failed write may result in denial of service for the end user. All requests issued to data store fail when it is unavailable due to either hardware or software failures, natural disasters, power outages, human errors, and others. It is possible for a subset of data store requests to fail when the data store is either sharded or offered as a service. With a sharded data store, the failed requests may reference shards that are either unavailable or slow due to load imbalance and background tasks such as backup. With data store as a service, requests fail when the application exhausts

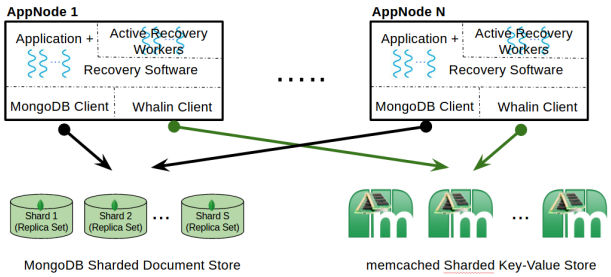


Figure 4.2: CADS architecture.

its pre-allocated capacity. For example, with the Amazon DynamoDB and Google Cloud Datastore, the application must specify its read and write request rate (i.e., capacity) in advance with writes costing more [6, 33]. If the application exhausts its pre-specified write capacity then its writes fail while its reads succeed. Using the proposed TARDIS techniques, an application may process failed writes using its cache. Note that cache misses of Table 4.2 no longer return an error "unavailable" for TARD and TARDIS. Both techniques read the data item from the data store, compute its latest value by merging with its buffered writes, insert the entry in the cache for future use, and service the request.

4.3 A Family of Techniques

Figure 4.2 shows a CADS consisting of Application Node (AppNode) servers that store and retrieve data from a data store layer, MongoDB. They use a caching layer for temporary staging of data [25, 63, 58, 24, 40]. This section presents TARDIS family of techniques using this architecture consisting of N multi-threaded AppNode server, a sharded data store, and a sharded memcached server.

Application thread (App thread) and Active Recovery (AR) worker refer to a thread of the AppNode server (see Figure 4.2). They implement the application

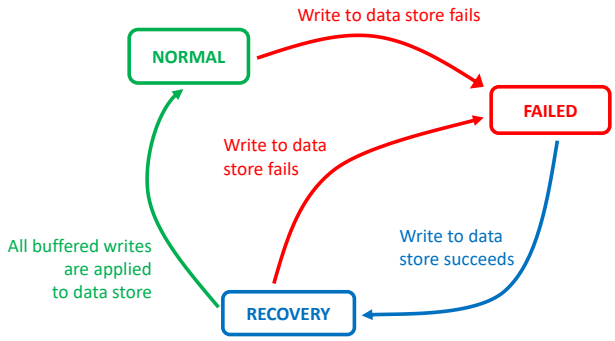


Figure 4.3: AppNode state transition diagram.

logic or a recovery worker that propagates buffered writes to the data store, respectively. We assume a failed request is due to the data store becoming unavailable completely. Section 4.3.5 discusses sharded data stores and data store as a service that may process some data store reads/writes while resulting in failed writes at the same time.

To simplify discussion and without loss of generality, we present the design of TARDIS since it is the most complex technique. While TAR, TARD and DIS are simpler, their implementation uses the principles of TARDIS outlined in this and the next section. This section presents the design with 1 multi-threaded AppNode. Section 4.4 extends the discussion to N multi-threaded AppNodes.

Table 4.5: List of Terms and Their Definitions.

Term	Definition
AR	Active Recovery worker applies buffered writes to the data store eagerly.
D_i	A data store document identified by a primary key P_i.
P_i	Primary key of document D_i. Also referred to as document id.
Δ_i	A key whose value is a set of changes to document D_i.
$\{K_j\}$	A set of key-value pairs associated with a document D_i.
M_j	A key that stores the mapping from K_j to Δ_i. It may also be stored alongside K_j.
TeleW	A key whose value contains P_i of documents with teleported writes.
ω	Number of TeleW keys.

To enhance performance, TARDIS does not prevent all undesirable race conditions. Instead, it detects their occurrences and resolves them. Pseudo-codes of this section highlight this design decision, e.g., see discussion of Step 6 of Algorithm 6.

Overview: Figure 4.3 shows the state transition diagram for the normal, failed, and recovery modes of TARDIS . TARDIS operates in normal mode as long as the data store processes writes in a timely manner. A failed data store write transitions AppNode to failed mode. In this mode, App threads buffer their data store writes in the cache. They maintain cached key-value pair impacted by the write as in normal mode of operation. Moreover, the AppNode starts one or more Active Recovery (AR) workers to detect when the data store is available to process writes again. Once an AR worker's write succeeds, the AR worker transitions AppNode state to recovery mode.

In recovery mode, TARDIS requires AppNode that issues a request that references a document with buffered writes to propagate the buffered write to the data store. This *lazy* technique may result in a long recovery duration when documents with buffered writes are not referenced by a request for a long time. AR workers expedite recovery by propagating buffered writes to the data store *eagerly*. The number of AR workers is a configurable parameter. A large number of AR workers propagate buffered writes to the data store at a higher rate. This may impose a high load on the data store and slow down AppNode's writes and reads (due to cache misses) to the data store. Once the AR workers apply all buffered writes to the data store, an AR worker switches AppNode to normal mode.

When either AppNode or an AR worker incurs a failed write, it switches AppNode mode from recovery to failed mode. A data store with an intermittent network connectivity may cause AppNode to toggle between failed and recovery modes repeatedly, see Figure 4.3.

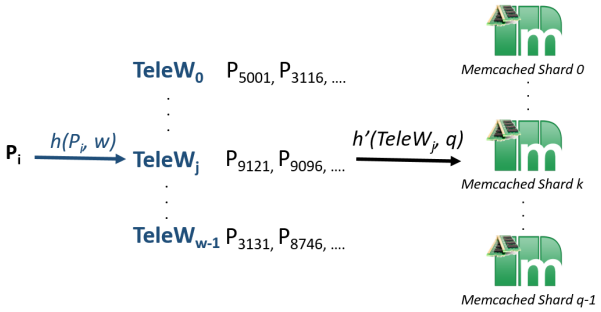


Figure 4.4: Documents are partitioned across ω TeleW keys. TeleW keys are sharded across q memcached servers.

TARDIS employs stateless leases such as Redlease[3] [7] to prevent undesirable race conditions between AppNode and AR workers. A lease is granted on a key to provide mutual exclusion for requests that reference this key. A lease has a fixed lifetime. Once a lease on a key is granted to a caller (say AppNode), the request by an AR worker must back off and try again. The duration of backoff is based on an exponential decay. The pending request is granted the lease once either the current request releases its lease or the lease expires.

Below, we detail normal, failed, and recovery modes in turn. This discussion uses the terminology of Table 4.5.

4.3.1 Normal Mode

In normal mode of operation, an AppNode thread processes a read action by looking up the value of one or more keys $\{K_j\}$ in the cache. If it is not found then this action has observed a cache miss. The AppNode thread queries the data store for the document D_i to compute the missing key-value pair(s) and stores them in the cache for future look up. Finally, it provides a response for the action.

[3]Redlease [7] can be implemented by AppNode with no changes to the memcached/Redis server.

With a write-action, we assume a write-through policy where an AppNode thread updates both the impacted data store document(s) and their corresponding cached key-value pairs.

Concurrent reads and writes of the same key-value pair may suffer from undesirable race conditions, causing the cache to produce stale data. Inhibit (I) and Quarantine (Q) leases of [31] prevent these undesirable race conditions. Moreover, they implement the concept of a session that provides atomicity at the granularity of multiple key-value pairs. These leases are detailed in Section 4.4.2.2.

4.3.2 Failed Mode

In failed mode, an AppNode thread processes a write for a data store document D_i with primary key P_i by generating a change, δ. The AppNode thread appends δ to the value of a key Δ_i in the cache, creating (Δ_i,δ) if it does not exist. The value of Δ_i is a chronological order of failed writes applied to D_i, $\{\delta_1, \delta_2, ..., \delta_j\}$. This list of changes is a *buffered write*.

Algorithm 5: AppNode in Recovery Mode (P_i)

```
1 acquire lease P_i;
2 V ← get(P_i+dirty);
3 if V exists then
4 |   success ← Update D_i in the data store using its buffered writes;
5 |   if success then
6 |   |   delete(P_i+dirty);
7 |   |   delete Δ_i (if any);
8 release lease P_i;
```

In addition, with TARDIS (and TARD), the AppNode thread generates a mapping M_j from each referenced K_j to Δ_i, and appends P_i to the value of a key named Teleported Writes (*TeleW*). TeleW is also stored in the cache. It is used by AR

workers to discover documents with pending buffered writes. Note that Δ_i may be redundant if the application is able to construct document D_i using its representation as a collection of cached key-value pairs.

The value of keys $\{K_j\}$ in the cache may be sufficient for AppNode to reconstruct the document D_i in recovery mode and write it to the data store. However, assuming memory is not a limiting factor, a developer may also generate a list of changes $(\Delta_i, \{\delta\})$ for the document to expedite recovery. This optimization applies when it is faster to read and process Δ_i instead of reading the cached value of $\{K_j\}$ to update the data store. An example is a member P_i with 1000 friends. If in failed mode, P_i makes an additional friend P_k, it makes sense for AppNode to both update the cached value and generate the change δ=push(P_k,Friends). In recovery mode, instead of reading an array of 1001 profile ids to apply the write to document D_i, the system reads the change and applies it. Since the change is smaller and its read time is faster, this expedites recovery.

In failed mode, AR workers try to apply buffered writes to the data store. An AR worker identifies these documents using the value of TeleW key. Each time AR's data store write fails, the AR waits for some time before repeating the write with a different document. This delay may increase exponentially up to a maximum threshold. Once a fixed number of AR writes (say p) succeeds, an AR worker transitions the state of AppNode to recovery.

TARDIS prevents contention for TeleW and one cache server by maintaining ω TeleW key-value pairs. It hash partitions documents across these using their primary key P_i, see Figure 4.4. Moreover, it generates the key of each ω TeleW with the objective to distribute these keys across all cache servers. In failed mode, when the AppNode generates buffered writes for a document D_i, it appends the document to the value of the TeleW key computed using its P_i.

A TeleW key may contain duplicates of P_i since a write that impacts D_i appends P_i to the relevant TeleW. TARDIS may represent TeleW as a hash set if the cache server supports it, e.g., Redis. TARDIS may also periodically compacts TeleWs to remove duplicates.

4.3.3 Recovery Mode

In recovery mode, TARDIS employs AR workers eagerly and AppNode lazily to apply buffered writes to those documents identified by TeleW. With AppNode, each time a user action references a document D_i with buffered writes, AppNode applies its writes to the data store prior to servicing the user action. In essence, during recovery, AppNode stops producing buffered writes and propagates buffered writes to the data store.

Challenges of implementing TARDIS's recovery mode are two folds:

1. Correctness: TARDIS must propagate buffered writes to documents in a consistent manner.

2. Performance: TARDIS must process AppNode's reads an writes as fast as normal mode while completing recovery quickly.

Correctness: We categorize buffered writes into idempotent and non-idempotent. Idempotent writes can be repeated multiple times and produce the same value. Non-idempotent writes lack this behavior. An example idempotent write is to change the value of a property, say name, of a document to a new value. A non-idempotent change is to add a photo to an album. Repeating this non-idempotent change N times results in an album with N copies of the same photo. However, repeating the idempotent change N times is side-effect free.

In recovery mode, multiple AR workers may perform idempotent writes multiple times without compromising correctness. However, this redundant work slows down the data store for processing regular requests and makes the recovery duration longer than necessary. When AppNode performs a write action using a document with buffered idempotent writes, an AR worker should not apply the same idempotent write again. This undesirable race condition loses the AppNode's write action.

Our implementation of recovery uses leases to apply a buffered write to a document once, either by an AR worker or AppNode. We assume the likelihood of AppNode's requests referencing a document with buffered writes is small in recovery mode. While our design is functional when this assumption is violated, it may be possible to develop more efficient designs per discussions at the end of this section.

In recovery mode, to perform an action that references a key K_j with a document D_i, if M_j exists, AppNode applies its buffered writes Δ_i to the data store. It does so by obtaining a lease on P_i, see Algorithm 5. Once the lease is granted, it looks up Δ_i. If it does not exist then it means an AR worker (or another AppNode thread) competed with it and propagated D_i's changes to the data store. In this case, it releases its lease and proceeds to service user's action. Otherwise, it applies the buffered writes to update D_i in the data store. If this is successful[4], AppNode deletes Δ_i to prevent an AR worker from applying buffered writes to D_i a second time.

Performance: To enhance performance of user issued requests, AppNode does not update the value of TeleW keys after updating D_i in the data store. This task is left for the AR workers as detailed next.

[4]This fails if the data store is unavailable.

Algorithm 6: Each Iteration of AR Worker in Recovery Mode

```
1  Initialize R ← {};
2  T ← A random value between 0 and ω;
3  V ← get(TeleW_T);
4  {D} ← Select α random documents from V;
5  for each D_i in {D} do
6      V ← get(P_i+dirty);
7      if V exists then
8          acquire lease P_i;
9          V ← get(P_i+dirty);
10         if V exists then
11             success ← Update P_i in the data store with buffered writes;
12             if success then
13                 Delete(P_i+dirty);
14                 Delete Δ_i (if any);
15                 R ← R ∪ P_i;
16         else
17             R ← R ∪ P_i;
18         release lease P_i;
19     else
20         R = R ∪ P_i;
21 if R is not empty then
22     acquire lease TeleW_T;
23     Do a multiget on TeleW_T and all P_i+dirty in R;
24     V ← Value fetched for TeleW_T;
25     For those P_i+dirty with a value, remove them from R;
26     Remove documents in R from V;
27     put(TeleW_T,V);
28     release lease TeleW_T;
```

Algorithm 6 shows each iteration of AR worker in recovery mode. An AR worker picks a TeleW key randomly and looks up its value. This value is a list of documents written in failed mode. From this list, it selects α random documents $\{D\}$. For each document D_i, it looks up Δ_i to determine if its buffered writes sill exist in the cache. If this key exists then it acquires a lease on D_i, and looks up

Δ_i a second time. If this key still exists then it proceeds to apply the buffered writes to D_i in the data store. Should this write succeed, the AR worker deletes Δ_i and buffered writes from the cache. Section 4.6.3 analyzes the value of α and its impact on recovery duration.

The AR worker maintains the primary key of those documents it successfully writes to the data store in the set `R`. It is possible for the AR worker's write to D_i to fail. In this case, the document is not added `R`, leaving its changes in the cache to be applied once the data store is able to do so.

An iteration of the AR worker ends by removing the documents in `R` (if any) from its target TeleW key, Step 6 of Algorithm 6. Duplicate P_is may exist in TeleW$_T$. A race condition involving AppNode and AR worker generates these duplicates: a write generates new buffered writes for a document between Steps 5 and 6 of Algorithm 6. Step 6 does a multi-get for TeleW$_T$ and the processed Δ_i values. For those P_is with a buffered write, it does not remove them from TeleW$_T$ value because they were inserted due to the race condition.

4.3.4 Cache Server Failures

A cache server failure makes buffered writes unavailable, potentially losing the content of volatile memory altogether. To maintain the availability of buffered writes, TARDIS replicates buffered writes across two or more cache servers. Our implementation with both Redis [2] and memcached [3] support this feature.

A cache server such as Redis may also persist buffered writes to its local storage, disk or flash [1]. This enables buffered writes to survive failures that destroy the content of volatile memory. Otherwise, buffered writes become unavailable when a cache server fails. During this time, to preserve consistency, AppNode in recovery mode may not process reads and writes since there is a potential buffered write

stored in the failed cache server. This possibility prevents processing of reads and writes even when the data store is available. This requirement is eliminated by replicating buffered writes across multiple cache servers to render them available in the presence of cache server failures.

4.3.5 Discussion

In failed mode, an AppNode thread does not perform writes to the data store that result in repeated timeouts. Instead, it maintains a volatile boolean variable, DataStoreAvailable, that is shared by AppNode threads and AR threads. When DataStoreAvailable is true, it means the data store is available and AppNode performs writes using it. Otherwise, the data store is unavailable and AppNode generates buffered writes. AppNode sets DataStoreAvailable to false when it detects a failed write. An AR worker sets DataStoreAvailable to true when it succeeds in recovering a document by writing it to the data store. Use of DataStoreAvailable is orders of magnitude faster than timing out by writing to a failed data store. We do not use atomic variables or protect the value of DataStoreAvailable using latches. In the presence of a race condition and in the worst case scenario, the AppNode threads perform a few more failed writes than necessary. In our experiments, implementing DataStoreAvailable using Java volatile provides a throughput that is 3% to 12% higher than using Java atomic.

Algorithm 5 does not show several important details of the AR. First, in Step 5, every time the request for a lease on P_i backs off, the AR worker abandons P_i and moves to the next document in $\{D\}$. This is because either another AR worker or AppNode is recovering D_i and it is unreasonable for this worker to wait for the lease only to discover that buffered writes for D_i have already been applied (by not finding Δ_i in the cache).

Second, every time an AR worker detects an empty TeleW$_T$ key, it fetches the value of TeleW$_{(T+1)\%\omega}$. If all ω TeleW keys have an empty value, the AR worker attempts to acquire a lease on all TeleW keys. If one of its ω leases is denied, the AR worker does not back off and try again. Instead, it releases all its TeleW leases, sleeps for a random amount of time between a minimum and a maximum threshold, and tries to obtain its ω leases from the start. Once the AR worker has its ω leases on ω TeleW keys, it looks up the value of these TeleW keys in the cache. If they have no value or their value is the empty set then the AR worker transitions the system to normal mode of operation by setting the value of DataStoreAvailable to true. Next, it releases all its ω leases.

With a sharded data store, one or more AppNode threads may observe failed writes by referencing a shard that is overloaded. This transitions AppNode to failed mode. However, the AR worker may immediately apply the buffered write to that shard and succeed, transitioning the system to recovery mode and normal mode quickly. This may cause some AppNode threads to operate in different modes; potentially all different threads may exhibit all three modes at the same time. The pseudo-code of Algorithms detailed in this section is robust enough to preserve the correctness property. Moreover, performance observed by AppNode threads in failed mode is faster than normal mode. In recovery mode, an AppNode thread may be slower when it propagates the buffered write to the data store, see Section 4.6.3.

Note that the failed mode of operation is for writes only. Reads are beyond the focus of this study. For our purposes, we assume AppNodes process a cache miss by querying the data store always. Once the data store fails, the system may exercise alternatives including application specific solutions such as providing stale data or displaying advertisements.

With the data store as a service, the writes may fail even though the reads succeed because AppNode has exhausted its provisioned write capacity. In this case, the writes are buffered in the cache and the AR worker succeeds in applying them once the AppNode's write capacity is restored. It is possible to design intelligent techniques that prioritize writes, requiring TARDIS to perform the high priority ones using the data store while buffering the low priority ones. TARDIS may propagate these buffered writes to the data store periodically while ensuring all high priority writes are processed successfully. This technique is a future research direction, see Chapter 6.

4.4 TARDIS with N AppNodes

TARDIS of Section 4.3 supports multiple AppNodes when requests are partitioned such that a document is referenced by one AppNode. Otherwise, it results in undesirable race conditions that compromise application correctness. This happens when AppNodes have different mode of operations for PStore. This section presents the undesirable race conditions and two alternative designs for TARDIS to prevent these race conditions.

4.4.1 Undesirable race conditions

With multiple AppNodes and intermittent failed PStore writes, different AppNodes may operate in different modes. While AppNode 1 operates in failed or recovery mode, AppNode 2 may operate in normal mode[5]. This results in a variety of

[5]A lightly loaded system dominated by reads such that AppNode 2 processes reads with 100% cache hit while AppNode 1 observes a failed write.

undesirable read-write and write-write race conditions when user requests for a document are directed to AppNode 1 and AppNode 2.

To illustrate an undesirable write-write race condition, consider a user Bob who creates an Album and adds a photo to the Album. Assume Bob's "create Album" is directed to AppNode 1 for processing. If AppNode 1 is operating in failed mode then it buffers Bob's Album creation in the cache. Once PStore recovers and prior to propagating Bob's album creation to PStore, Bob's request to add a photo is issued to AppNode 2. If AppNode 2 is in normal mode then it issues this write to an album that does not exist in PStore.

This write-write race condition would not have occur if Bob's requests were directed to AppNode 1: AppNode 1 would have recovered Bob's pending Album write prior to performing the photo add operation, see Section 4.3.

An undesirable read-write race condition is when Bob changes the permission on his profile page so that his manager Alice cannot see his status. Subsequently, he updates his profile to show he is looking for a job. Assume Bob's first action is processed by AppNode 1 in failed mode and Bob's profile is missing from the cache. This buffers the write as a change (Δ) in the cache. His second action, update to his profile, is directed to AppNode 2 (in normal mode) and is applied to PStore because it just recovered and became available. Now, before AppNode 1 propagates Bob's permission change to PStore, there is a window of time for Alice to see Bob's updated profile.

The presented read-write race condition requires a cache miss. If Bob's profile was in the cache then it would have been updated by AppNode 1, preventing Alice from seeing his updated status.

4.4.2 Two Solutions: Recon and Contextual

One approach to address the undesirable race conditions of Section 4.4.1 is to require all AppNodes to have the same PStore mode of operation. This results in unnecessary complexity and is expensive to realize with a large number of AppNodes. Moreover, it does not work when the PStore is partially available.

We presents two solutions that do not require consensus among AppNodes. The first combines the normal and recovery mode into one, resulting in each AppNode to operate in two possible modes: recon (combined recovery and normal) and failed. The second employs stateful leases. We describe them in turn.

4.4.2.1 Solution 1: Recon

Recon combines recovery and normal mode of operation into one by requiring:

- Cache hits to be processed in the same manner as the normal mode of Section 4.3.1.
- All cache misses and application writes are processed in recovery mode. They must look up P_i+dirty to discover if their referenced document has buffered write. If so, they apply the change to PStore prior to processing the request.

Recon avoids the write-write race condition of Section 4.4.1 by requiring all writes to operate in recovery node. In our example, Bob's photo add operation is a write action and it is forced to recover Bob's pending write that creates the album prior to adding the photo.

Similarly, Recon avoids read-write race condition of Section 4.4.1 by requiring Alice's cache miss for Bob's profile to process pending writes for Bob's document. Once these changes are applied, Alice is not able to see Bob's change of status.

Recon is simple to implement. Its overhead is during normal mode of operation, namely the requirement for all cache misses and write actions to perform additional work than necessary by looking up P_i+dirty. This overhead is marginal for those applications whose workload exhibits a high ratio of reads to writes [16] with a high cache hit rate.

4.4.2.2 Solution 2: Contextual Leases

TARDIS with contextual leases (Contextual for short) maintains the three mode of operation described in Section 4.3. It incorporates I and Q leases of [31] and extends them with a marker to prevents the undesirable race conditions of Section 4.4.1. Below, we provide an overview of the I and Q leases as used during normal mode. Subsequently, we describe our extensions with a marker.

The framework of [31] requires: 1) a cache miss to obtain an I lease on its referenced key prior to querying the PStore to populate the cache, and 2) a cache update to obtain a Q lease on its referenced key prior to performing a Read-Modify-Write (R-M-W) or an incremental update such as `append`. Leases are released once a session either commits or aborts. Given an existing I or Q lease on a key, a request for an I or a Q lease for the same key is as follows:

Requested Lease	Existing Lease	
	I	Q
I	Backoff	Backoff
Q	Void I & grant Q	Deny and abort

If there is a request for an I lease on a key with an existing I lease then the requester must backoff and try again. This avoids thundering herds by requiring only one out of many requests that observes a cache miss to query PStore and

populate the cache. Ideally, all other requests should observe a cache hit in their retry.

If there is a request for a Q lease on a key with an existing I lease then there is a potential read-write race condition. The Q lease voids the I lease, preventing the reader from populating the cache with a value. This prevents the reader from inserting a stale value (computed using a the state of PStore prior to write) in the cache.

If there is a request for an I lease on a key with an existing Q lease then, once again, we have a potential read-write race condition. The I lease backoffs and tries again until the writer commits and releases its Q lease.

Finally, if there is a request for a Q lease on a key with an existing Q lease then we have a write-write race condition. In this case, the requester is aborted and restarted, forcing the R-M-W of one write to observe the R-M-W of the other. It is acceptable for the Q lease requester to back off and try again if a R-M-W action references only one key - however, as noted in Section 4.3, a write may R-M-W a collection of keys $\{K_i\}$.

In failed mode, a write action continues to obtain a Q lease on a key prior to performing its write. However, at commit time, a Q lease is converted to a P marker on the key. This marker identifies the key-value pair as having buffered writes. The number of keys with such markers increases monotonically as they are written in failed mode.

The P marker serves as the context for the I and Q leases granted on a key. It has no impact on their compatibility matrix. When the AppNode requests either an I or a Q lease on a key, if there is a P marker then the AppNode is informed of this marker should the lease be granted. In this case, the AppNode must recover the PStore document prior to processing the action.

To explain why Contextual prevents the undesirable race conditions of Section 4.4.1, observe: 1) there is at most one lease on a key, either I or Q, 2) an I lease is requested when a read action observes a cache miss, and 3) a Q lease is requested by a write action. When there is a buffered write for a document, either an I or a Q lease on its P_i encounters a P marker that causes the requesting AppNode to propagate the buffered write to the document in PStore. In essence, the mode of operation is at the granularity of a key.

In our example undesirable write-write race condition, Bob's Album creation using AppNode 1 generates a P marker for Bob's Album. Bob's addition of a photo to the album (using AppNode 2) must obtain a Q lease on the album that detects the marker and requires the application of buffered writes prior to processing this action. Similarly, with the undesirable read-write race condition, Bob's read (performed by AppNode 2) is a cache miss that must acquire an I lease. This lease request detects the P marker that causes the action to process the buffered writes. In both scenarios, when the write is applied to PStore and once the action commits, the P marker is removed as a part of releasing the Q leases.

4.5 Non-idempotent Buffered Writes

A buffered write may be a non-idempotent change to a document. For example, it may increment a property or insert a value in an array property of a document. In recovery mode, when an AppNode thread or an AR worker applies a buffered write to the data store, it may encounter a timeout (a failed write) even though the write succeeded. An obvious solution is to repeat the operation. While

this is acceptable with idempotent buffered writes, it is not acceptable with non-idempotent buffered writes because applying them two or more times may violate read-after-write consistency.

One possible approach is to extend the data store with application specific constraints that prevent inconsistent states due to multiple applications of a buffered write. For example, with a change that pushes a value into an array property of MongoDB and advanced knowledge that the array may not contain duplicate values, one may specify a constraint that requires the values in an array property to be unique. This prevents multiple applications of a non-idempotent buffered write from violating consistency. Such approaches may not be possible with all persistent data types and their use by an application. This section presents SnapShot Recovery (SSR) to address this challenge.

SSR, see Algorithm 7, is used in recovery mode when applying a buffered write to the data store. Its essence is to convert a buffered non-idempotent write for D_i into an idempotent buffered write by generating a snapshot of D_i. This snapshot is written to the cache and deleted immediately if the application of buffered write to the data store succeeds. It is used to detect failures in recovery mode and to apply a non-idempotent write once. Algorithm 7 is robust enough to enable a (different) AppNode to operate in failed mode and generate non-idempotent changes concurrently.

Details of SSR are as follows. The value of Δ_i includes a unique id (uid) generated[6] by AppNode when it creates this key for a failed write of D_i the very first time. In Steps 13, SSR reads the document D_i in memory and applies the non-idempotent changes to it to create the snapshot. In Steps 16-19, SSR initializes the snapshot S_i key-value pair and inserts it in the cache. Next, it applies the

[6]An integer counter concatenated with the mac address of AppNode server.

Algorithm 7: Snapshot Recovery (P_i)

```
1  acquireLease(P_i)
2  S_i ← Get cached snapshot of D_i using P_i
3  δ_i and uid ← Get Δ_i
4  if S_i ≠ null then
5      if δ_i = null or S_i.uid ≠ uid then
6          D_i ← Get document P_i from the data store
7          if D_i ≠ S_i.data then
8              if Overwrite D_i with S_i.data = fail then
9                  releaseLease(P_i)
10                 return fail
11     Delete S_i
12 if δ_i ≠ null then
13     D_i ← Get document P_i from the data store
14     D_i ← Apply δ_i to D_i
15     S_i ← Init snapshot
16     S_i.data ← D_i
17     S_i.uid ← uid
18     Store S_i in the cache
19     Delete Δ_i
20     if Apply δ_i to the data store = fail then
21         releaseLease(P_i)
22         return fail
23     Delete S_i
24 releaseLease(P_i)
25 return success
```

buffered write to D_i in the data store and deletes the snapshot S_i. Note that the snapshot maintains the uid of Δ_i.

When an AppNode thread or an AR worker applies a buffered write to D_i, SSR looks up its snapshot first (see Step 3). If it exists then SSR has detected a failure when a buffered write was being applied to the data store. Hence, it fetches D_i from the data store and compares it with S_i. If they are not equal then SSR

over-writes D_i with S_i and deletes S_i. Subsequently, it applies new buffered writes (if any).

Note that the writing of D_i to the data store, Steps 8 and 20, may encounter a failed write. Hence, SSR returns an error to the calling thread, either AppNode or AR worker. If it is an AR worker, this thread may invoke SSR immediately or after some delay. Alternatively, it may ignore the failure and let another AR thread tries it at a later time. An AppNode thread encounters this failure when performing a read or a write. With a read, it may ignore the failure and consume the value from the cache, deferring to an AR worker to apply the buffered write to the data store. Similarly, with a write, an AppNode thread may defer to an AR worker by generating a buffered write.

The uid is used when there is a failure after Step 19 and before Step 23, e.g., the condition of Step 20 is satisfied. In this case, the buffered write Δ_i is no longer present in the cache. However, its snapshot S_i is available. Should an AppNode thread in failed mode create a buffered write Δ_i then it will a uid different than S_i's uid. Steps 6-11 overwrite the document D_i with S_i. Subsequently, SSR applies the new Δ_i to this D_i.

The *overhead* of SSR is writing and deleting the snapshot of a document impacted by a buffered write prior to its application to the data store, see Steps 18 and 23. This overhead is incurred in recovery mode.

4.6 Evaluation

We used YCSB [19] and BG [13] to evaluate TARDIS. We focused on YCSB's update heavy workload A consisting of 50% read and 50% update. With BG, we used its moderate read heavy workload of 100 reads for every 1 write. It consists

of the following mix of interactive social networking actions: 89% View Profile, 5% List Friends, 5% View Friend Request, 0.4% Invite Friend, 0.2% Accept Friend Request, 0.2% Reject Friend Request, and 0.2% Thaw Friendship.

We evaluated the following cache managers: Redis 3.2.8, CAMP 1.0, Twemcached 2.5.3, memcached 1.4.36 with and without chunked version configuration. The later supports key-value sizes greater than 1MB. CAMP is a variant of memcached with two changes. First, it uses malloc and free to manage memory space instead of a slab implementation. Second, it uses the CAMP replacement algorithm [28] that is cost aware instead of LRU.

Variants of memcached implement the same commands. Redis provides richer commands and data types along with server scripting, resulting in a different implementation. Given a fixed amount of cache size, different caches may store a different number of buffered writes.

The physical representation of buffered writes with YCSB is different from BG because their writes are different in nature. A YCSB update overwrites values of several properties of a document. A BG write action such as “Invite Friend" pushes the member id of the inviter into the invitee’s list of pending friend invitations[7]. Section 4.6.1 details representation of buffered writes with YCSB and BG.

We extended the workload generator of YCSB and BG to implement Recon and Contextual. With both implementations, the same physical YCSB and BG database organizations are used in MongoDB and a cache manager. Both benchmarks databases consist of 100,000 MongoDB documents. Moreover, both are configured to generate a skewed pattern of data access using a Zipfian distribution. In the reported experiments, we emulate a fixed duration of failed writes by

[7]This is represented as an array data type.

terminating the mongod process of MongoDB and restarting it after the specified duration.

In these experiments, a multi-node workload generator emulates multiple AppNodes. These are connected to a server hosting MongoDB version 3.2.9 as our data store and a second server hosting Twemcache version 2.5.3 as our cache server. Each server is an off-the-shelf PC configured with 4-Core Intel i7-3770 3.40 GHz CPU 16 GB of memory and 1 gigabit per second (Gbps) networking card.

We observe Recon and Contextual have comparable performance characteristics and present the evaluation results of Contextual.

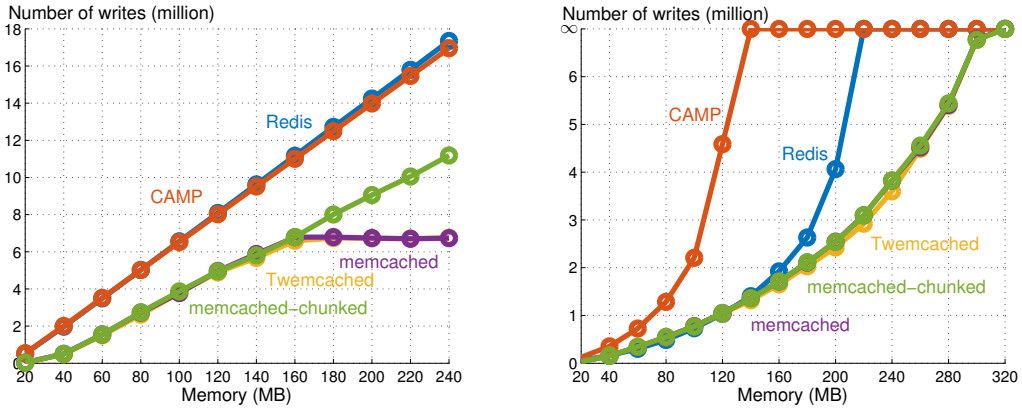
4.6.1 Physical Design of Buffered Writes

A buffered write with both benchmarks is a key-value pair. The key Δ_i identifies a unique MongoDB document impacted by a write. Its value is a buffered write.

With Redis, a buffered write for YCSB is a hash map consisting of a pairing of the property name with the updated value. Hence, there is a maximum bound on the size of a buffered write with YCSB. It is defined by the number of properties of a document (10). Moreover, generation of a buffered write is one network round-trip that sets the property:value pairings in the hash map.

Variants of memcached do not support a hash map data type. Hence, we represent a YCSB buffered write as a list of property:value pairings. Generation of this buffered write performs two network round-trips by implementing a read-modify-write operation.

With BG, a buffered write is a list of logical operations such as a push or a pop of a value from an array property of MongoDB document. This implementation is used with all cache managers. We make the logical operations idempotent by



(a) BG (b) YCSB

Figure 4.5: Number of writes processed by TARDIS as a function of the amount of available memory.

configuring the array property of MongoDB documents to contain unique values. Hence, this implementation does not use the SSR technique of Section 4.5.

The memory required for TARDIS to cache buffered writes depends on the characteristics of the workload. In Figure 4.5, we configure a cache manager with a fixed amount of memory (x-axis) and issue failed writes until the cache manager either refuses to insert the buffered write or evicts a buffered write. The y-axis of this figure is the number of buffered writes.

The YCSB results, Figure 4.5.b, show all cache managers are able to accommodate an infinite number of buffered writes beyond a certain memory limit. There are two reasons for this. First, the number of documents is fixed at 100,000. Second, the size of a buffered write for a document is fixed and *independent* of the number of failed writes performed on that document. Once all YCSB documents are referenced by a failed write, the memory requirement of TARDIS levels off.

The implementation of a YCSB buffered write with CAMP, Twemcached, and memcached are identical and have the same size. This size is more compact than the hash map of Redis. Both CAMP and Redis use malloc and free to manage

the available space. CAMP reaches infinity with a lower memory because it caches small key-value pairs by evicting larger ones. Twemcached and memcached manage memory space using a slab implementation. This implementation suffers from calcification [43, 40], requiring more memory to accommodate an infinite number of failed writes.

	Uniform		Zipfian	
	0 AR	20 AR	0 AR	20 AR
TAR/TARD	∞	∞	∞	∞
DIS	96	14	600	12
TARDIS	45	14	307	12

Table 4.6: Recovery Duration (seconds) with a write-heavy workload (50% writes).

With BG, the size of a buffered write for a document increases monotonically as a function of the number of failed writes referencing the document. There is no memory size that accommodates an infinite number of writes, see Figure 4.5. Redis and CAMP support approximately the same number of writes because the size of a buffered write is the same with all cache managers. Twemcached and memcached variants continue to suffer from slab calcification. Both Twemcached and memcached cannot support additional writes beyond 160 MB of memory because the maximum key-value size is 1 MB. Chunked variant resolves this limitation to utilize memory capacities higher than 160 MB.

4.6.2 Performance of TARDIS Family of Techniques

This section evaluates TAR, TARD, DIS, and TARDIS. Our objective is to highlight their tradeoffs and identify important factors that dictate both the recovery duration and impact on system performance (foreground tasks) in recovery mode. The main lessons of this evaluation are:

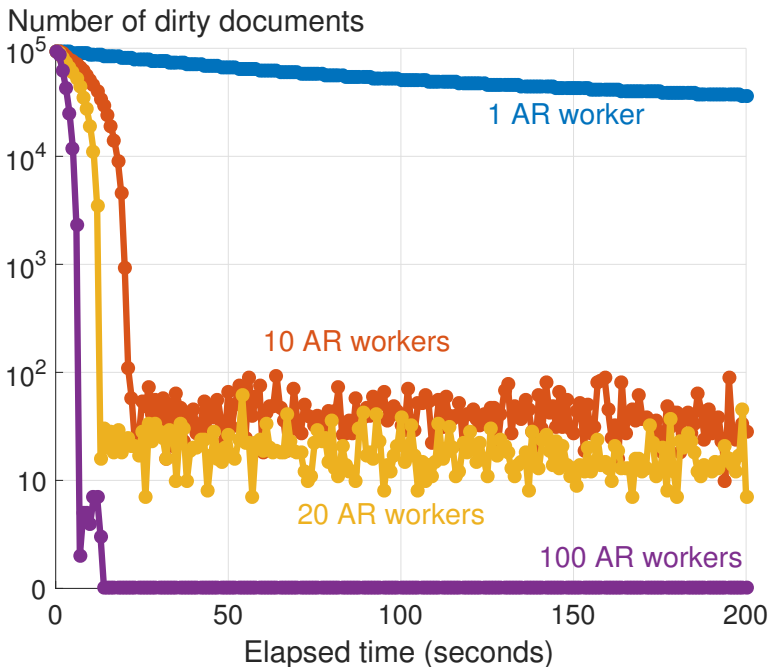


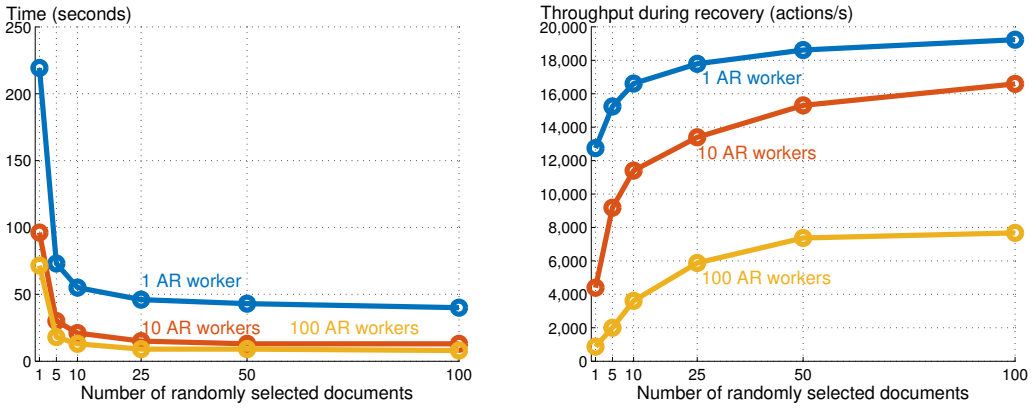
Figure 4.6: Impact of AR workers on the number of dirty documents with TAR and 3% write.

- Recovery duration is dictated by the number of AR workers, foreground load, and the frequency of writes. A higher number of AR workers applies buffered writes to the data store at a higher rate. They compete for resources with the foreground load and slow down their rate of buffered writes production.

- Increasing number of AR workers expedites recovery for all techniques, especially with a skewed pattern of access to the data.

- DIS and TARDIS are the fastest techniques to complete recovery. TAR and TARD may complete recovery depending on the number of AR workers, i.e., a higher number slows down the foreground load.

We demonstrate these lessons, in turn, using the YCSB benchmark. We stage 10,000 documents in the cache. Next, we fail the data store and impose a write-only workload that updates 95,000 documents, i.e., 95% of the YCSB database.



(a) Recovery duration

(b) Throughput during recovery

Figure 4.7: TARDIS Recovery with YCSB as a function of the number of AR workers and the number of randomly selected documents (α) in each iteration.

For those 10,000 documents that are in the cache, each write updates the cache entry (write-through) and generates the buffered write. For those 85,000 missing from the cache, each write generates the buffered write only. Next, we recover the data store and issue a foreground load consisting of 20 threads issuing YCSB workload A consisting of 50% read and 50% write. This emulates recovery mode and it ends once all buffered writes are applied to the data store.

Rows of Table 4.6 show the recovery duration with different techniques. Columns of this table pertain to two different access patterns to the documents, uniform and skewed. For each, we report the recovery duration with 0 and 20 AR workers. Table 4.6 shows TAR and TARD do not complete recovery with 0 and 20 AR workers. DIS and TARDIS complete recovery with no AR workers. Twenty AR workers expedite their recovery by several orders of magnitude.

DIS is 50% slower than TARDIS because it uses half the foreground load (writes) to apply buffered writes to the data store. TARDIS uses both writes and reads that observe cache misses to propagate buffered writes to the data store, expediting recovery.

Even with 20 AR workers, TAR and TARD fail to complete recovery. In these experiments, TARâĂŹs cache hit rate is a constant 6% because cache misses are not processed even though the data store is available. In contrast, TARD provides a 94% cache hit rate by requiring cache misses to apply their buffered writes to the data store prior to computing the missing cache entry.

In order for TAR and TARD to complete recovery, the workload must produce writes at a slower rate. One way is to increase the number of AR workers to compete with the foreground threads issuing writes. Another way is to realize this is to reduce the frequency of writes in the workload. Figure 4.6 shows the number of dirty documents with TAR and 3% write using a different number of AR workers. This figure shows TAR completes the recovery duration with 100 threads after 13 seconds. With 1 AR worker, its number of dirty documents is in the order of tens of thousands. This number is reduced to less than 100 with 10 and 20 AR workers.

TAR and TARD may propagate buffered writes to a document only to have a foreground write generating a buffered write for that document. This means buffered writes for a document must be applied to that document several times. For example, with TAR and YCSB workload A (50% write) using a skewed access pattern, AR workers recover 99% of unique documents by applying 1,559,050 buffered writes[8]. This means, on average, more than ten buffered writes are applied to every YCSB document. With a skewed pattern of access, a significantly larger number of buffered writes was applied to a single document.

While TARDIS expedites recovery to transition the system to normal mode sooner, when compared with TAR, it requires extra memory from the cache server to maintain mappings of Section 4.3. (It is also more complex to implement than

[8]This is 10 minutes into recovery mode.

TAR as detailed in Section 4.1. In our experiments, the mapping is the identity function consisting of a key that identifies the document (4 bytes) and a 1-byte flag. These mappings occupy 464 KB of memory, i.e., 95,000 written documents multiplied by 5 bytes per document at the start of the experiment.

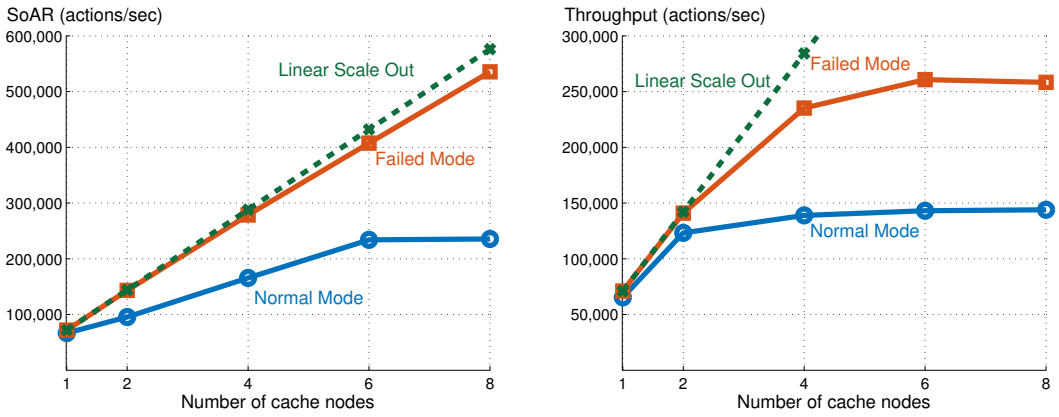
4.6.3 Recovery Duration

We use YCSB and BG to analyze recovery duration by varying the number of AR workers and the number of documents selected randomly by each worker, α, see Figure 4.7.a. In these experiments, the number of TeleW keys is set to 211 and assume a low system load of 20 threads issuing requests during recovery.

We observe that the recovery duration reduces an order of magnitude from 1 AR worker to 10 AR workers across all alpha values. A large number of AR workers with a small α value is superior to having a few AR workers with a high α value. 100K buffered documents (due to a 20-minute failure) are recovered in less than 10 seconds with a large number of AR workers (i.e., 10 and 100) and alpha value of 50. This is at the expense of a lower throughput observed with 20 threads issuing requests, see Figure 4.7.b.

TARDIS's recovery duration is faster with BG because its workload is read heavy. Its contention for the data store is lower than YCSB, enabling AR workers to complete recovery faster. However, the observed trends are similar to those of YCSB.

Results of this section highlight the following trade-off. A faster recovery duration is at the expense of a lower observed performance by the foreground requests. The number of AR workers is a knob that enables an application to control this tradeoff.



(a) BG (CAMP) (b) YCSB (Redis)

Figure 4.8: TARDIS horizontal scalability.

4.6.4 Horizontal Scalability of TARDIS

A key metric of BG is SoAR, the throughput provided by a system while satisfying a pre-specified service level agreement (SLA). The y-axis of Figure 4.8a.a shows SoAR of TARDIS as a function of the number of cache servers, x-axis. The SLA requires 95% of actions to observe a response time faster than 100 milliseconds. We show the observed SoAR in both normal mode of operation with TARDIS and failed mode when TARDIS generates buffered writes. Figure 4.8a.b shows the scalability of the caching layer relative to the SoAR observed with one cache server.

Obtained results show TARDIS performance and scalability in failed mode is superior to normal mode of operation. In failed mode, the framework scales almost linearly as a function of the number of cache servers. However, the system struggles to scale in normal mode, with performance leveling off beyond 6 servers. In normal mode, TARDIS performs a write action using both the cache and MongoDB. The server hosting MongoDB is the bottleneck limiting performance. Even with one cache server, the CPU of the server hosting MongoDB spikes from 20% to 100%.

Beyond 4 cache servers, the CPU of the server hosting MongoDB becomes 100% utilized. This causes the throughput to level off.

In failed mode, all writes insert buffered writes in the caching layer because MongoDB is not available. These operations are hash partitioned using the document id, enabling the system to scale almost linearly. It is not perfectly linear due to temporary bottlenecks caused by concurrent requests colliding and referencing the same cache server.

Horizontal scalability of the system with YCSB is limited because 50% of requests are updates, see Figure 4.8b. These results were obtained with 4 YCSB clients issuing requests. Each client is configured with 200 threads, for a total of 800 threads generating requests concurrently. Prior to updating a key-value pair, the YCSB client obtains a lease on the key-value pair. When two or more concurrent threads try to update the same key-value pair, one succeeds while others back-off and try again. This isolates and serializes concurrent updates. It also causes cache servers to sit idle and wait for work, reducing performance and resulting in sub-linear scalability. The failed mode is more efficient than the normal mode as it generates a buffered write instead of updating MongoDB. By processing a larger number of actions per second, it also observes more than 25% of requests to back-off than normal mode.

4.7 Proof of TARDIS Consistency

In this section, we provide a formal proof of read-after-write consistency for TARDIS. We assume (a) writes transition the database from one consistent state to another and (b) the data store provides strong consistency.

Definition 4.7.1. *A read R_i for a document D_i from the data store may be served by a key-value pair (k_i, v_i) from the cache, where k_i=D_i and v_i=the property values of the document D_i.*

Definition 4.7.2. *A write W_j impacts a document D_j if W_j inserts D_j, deletes D_j, or updates one or more properties of D_j.*

Corollary 4.7.2.1. *A write W_j for D_j impacts D_j in the data store and its corresponding (k_i, v_i) in the cache atomically.*

Definition 4.7.3. *A buffered write for a document D_i is represented as a key-value pair (k_i^{bw}, v_i^{bw}) where k_i^{bw} is "BW:DocID", v_i^{bw} stores all the pending changes of D_i that must be applied to the data store.*

Definition 4.7.4. *A* mapping *inputs the key k_i for a cache entry to compute the keys $\{k_i^{bw}\}$ of the buffered writes. Without loss of generality, we assume the function f is $f(DocID) \rightarrow \{k_{DocID}^{bw}\}$. Given the identity of a document, DocID, this function computes the key of buffered writes.*

A read R_i identifies its referenced key k_i and looks it up from the cache. If v_i exits, R_i consumes v_i. If v_i does not exist, the cache grants R_i an I lease. In *normal* mode, R_i queries the data store to compute v_i, puts (k_i, v_i) in the cache and releases the I lease. In *failed* mode, it releases the I lease and returns `READ_ERROR` to the application. The application may process this error as an exception. In *recovery* mode, the read acquires a Q lease on the buffered write key k_i^{bw} and looks up its value v_i^{bw} from the cache. If v_i^{bw} is found, the read applies it to the data store and releases the Q lease on k_i^{bw}. It then queries the data store to compute v_i, puts it in the cache and releases the I lease on k_i.

A write W_j identifies its impacted keys and acquires a Q lease on each key k_i. If value v_i of k_i exists then W_j constructs a copy v_i^c of v_i in the cache and updates v_i^c.

In *normal* mode, it updates the data store synchronously and releases its Q leases. In *failed* and *recovery* mode, it acquires Q leases on all buffered writes mapped to each of its impacted keys k_i. In failed mode, it incorporates its changes to each buffered write k_j^{bw} and releases the Q lease on it. In the recovery mode, it looks up the value v_j^{bw} for each k_j^{bw}, applies v_j^{bw} to the data store and releases the Q lease on it. It then updates the data store synchronously and releases the Q leases on its impacted keys.

Theorem 3. *In failed mode, TARDIS provides either read-after-write consistency for a read of D_i with buffered write(s) or returns an error.*

Proof. R_i looks up k_i in the cache. If v_i exists, it reflects the last write of D_i produced as output. Otherwise, R_i has observed a cache miss. In failed mode, R_i is unable to apply the buffered write v_i^{bw} (if any) to the data store or query the data store to compute v_i. Hence, it returns `READ_ERROR`. ■

Theorem 4. *In recovery mode, TARDIS provides read-after-write consistency while applying pending buffered writes to the data store.*

Lemma 4.1. *In recovery mode, TARDIS provides read-after-write consistency for a read of D_i processed concurrently with pending buffered writes for D_i being applied to the data store.*

Proof. If R_i observes a cache hit for k_i, it consumes its value v_i. Last write that released its Q leases on k_i produced the latest v_i. A concurrent write W_j creates a copy v_i^c of v_i and the read observes the original version v_i. This serializes R_i before W_j.

If R_i observes a cache miss, it looks up k_i^{bw} for pending writes. If another request also looks up k_i^{bw}, only one may proceed because each must acquire a Q lease on k_i^{bw}. The other aborts and retries. If v_i^{bw} is a non-idempotent change,

it is converted to an idempotent change, stored in the cache, and the Q lease is released. Subsequently, R_i obtains a Q lease on this idempotent change, applies it to the data store, deletes the entry and releases its lease. Hence, the buffered write is applied once to the data store and deleted from the cache. Subsequently, R_i queries the data store to compute v_i, puts it in the cache, and releases its I lease. R_i observes the latest value and is serialized after all writes that buffered their changes in v_i^{bw}. ■

Lemma 4.2. *In recovery mode, TARDIS maintains consistency of the database in the presence of arbitrary failures of threads that apply buffered writes to the data store.*

Proof. A thread that applies a buffered write to the data store may either be a read R_i or a TARDIS background thread. These threads change the state of the database. Failures of these threads cause their Q lease to time out, making the buffered write available again. The buffered write may either be idempotent or non-idempotent. If it is idempotent, its repeated application to the data store does not compromise database consistency. If it is non-idempotent, it has not yet been applied to the data store. A buffered write is always replaced with its idempotent equivalent prior to being applied to the data store. Hence, consistency of the database is preserved. ■

Lemma 4.3. *In recovery mode, TARDIS processes a write to the data store for D_i consistently in the presence of buffered writes for D_i.*

Proof. In recovery mode, a write always looks up the buffered writes $\{k_i^{bw}\}$ for its impacted key k_i. It acquires a Q lease on each buffered write in $\{k_i^{bw}\}$. If another concurrent thread is granted a Q lease on k_i^{bw}, this write aborts and retries. Otherwise, it applies v_i^{bw} to the data store, deletes (k_i^{bw}, v_i^{bw}) from the cache and

releases the Q lease. At this point, all changes of previous writes on D_i are reflected in the data store. Hence, this write may proceed to issue its write request to the data store. ■

Chapter 5

A CADS Framework

Chapter 3 shows that CADS enhances the system performance significantly with the write-back policy. The implementation of write-back relies on: i) a lease-based cache framework to enable the concept of sessions and support strong consistency, and ii) a resolution of mappings between SQL statements to buffered writes. In this chapter, we introduce two components that constitute the CADS framework:

- *LeCaF*. A lease-based cache framework that supports strong consistency. It extends the IQ-Framework [31] to enable the concept of sessions consisting of multiple key-value pairs.

- *NgCache*. A client-library that supports write-around, write-through and write-back policies. It is able to process writes asynchronously with the write-back policy by providing the developers with interfaces to declare the mappings, the buffered writes and how they are applied to the data store. NgCache uses LeCaF as a building block to issue requests to the cache servers.

This chapter is organized as follows. Section 5.1 describes LeCaF in details, including the implementation at both client and server sides. Section 5.2 describes NgCache and its interfaces.

5.1 LeCaF – A Leases-Based Cache Framework

A session consists of at most one data store transaction and multiple cache operations that are executed atomically. LeCaF uses sessions to maintain the data store and the cache consistent. A session may be a read or a write. A read session does not update any of its key-value pairs. If it observes a cache miss, it computes a key-value pair and sets it in the cache. A write session may perform read-modify-write or incremental update (for example, append and incr) on one or more cache entries.

A session may acquire leases on one or more keys. A lease [35] in distributed systems is a synchronization primitive that is similar to a lock in database systems. Different than a lock, a lease may expire. Leases serializes concurrent sessions, preventing undesirable race conditions to put stale data in the cache. With LeCaF, a lease may be a Shared (S) or eXclusive (X) and is granted at the granularity of a key. With write-through and write-back, details of these leases were discussed in Section 5.1.1.2. The lease compatibility table for write-around is different, as discussed in Section 5.1.1.1. A session may hold one or multiple leases on one or multiple keys during its life time. Once it commits or aborts, all leases granted to the session are deleted.

In our design, both the client and the server are aware of the existence of sessions. Each session has a unique session id. A coordinator is responsible to assign ranges of session ids to clients. When the client is initialized, it gets a range of session ids from the coordinator. When a session starts, a new unique session id is generated using the range. Once the range is exhausted, the client sends a request to the coordinator for a new range. The cache server are implemented with

data structures (session objects and lease objects, see Section 5.1.1) to keep track of sessions and their leases.

Section 5.1.1 describes the new commands at the cache server to support write-around, write-through and write-back policies. Section 5.1.2 describes the new methods implemented at the client to issue requests to the server utilizing the new server commands.

5.1.1 Server

The cache server maintains a session object for each session during its life time. The object includes the session id, the session's status and a mapping of keys involved in the session and their pending versions (if exist). A session status may be one of these values: *active*, *validated* or *aborted*. When a client initializes a session with the cache server, the cache server creates a session object, setting its state to *active*. The state changes to *validated* when the client validates the session successfully. Once validated, the server no longer may abort the session. The client commits the session by sending a commit message (LCommit()) to the server. Its session object is deleted at the server. Figure 5.1 shows the state transition of a session.

At any time, a client may abort its pending session with the LAbort() message. The session object is deleted from the server accordingly. A server may also abort a session when another session voids a S lease hold by it. In this case, the server does not delete the session object immediately. Instead, it sets the session state to be *aborted*. The client gets informed of the abort when it sends a subsequent request to the server and consequently deletes the session object.

A key has a lease object associated with it. A lease object has two fields: 1) *a lease type* (either Shared (S) or eXclusive (X)), and 2) *a list of session ids* holding

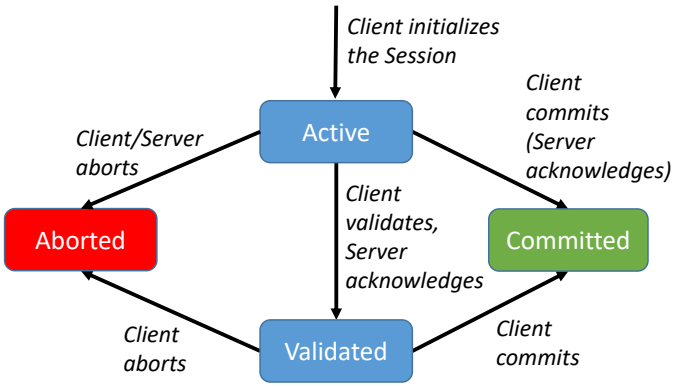


Figure 5.1: Session State Transition.

the lease. With write-through, because there is at most one session that may hold the X lease on a key, the list contains only one session id if the lease type is X.

We introduce new commands at the cache server in the context of sessions. Most commands extend existing commands to handle session and lease objects. These objects are transparent to the client. With a session, the client communicates with the server only by using the session id. When the server starts, it is configured with a specific cache write policy. Any attempts on issuing requests for commands belonging to another cache policy are replied with NOT_AVAILABLE. Below, we describe commands for each cache policy in details.

5.1.1.1 Write-Around

Requested Lease	Existing Lease	
	S	X
S	Grant S lease	Abort and Retry
X	Grant X and void S lease	Grant X lease

Table 5.1: S and X Lease Compatibility for Write-Around.

The lease compatibility for write-around is shown at Table 5.1. Different than write-through, write-around allows multiple sessions to hold for the same X lease on the same key. When the sessions holding the X lease commit, they delete the

key multiple times. This is acceptable since delete operation is idempotent. The commands for write-around policy are presented below.

`LGet(sess_id, key)`. The client uses this command to get the value. If the session object exists in the cache and its state is aborted, it is deleted and the server returns ABORT. Otherwise, there may have three possibilities:

1. *The key has no lease.* The server grants S lease on the key and returns VALUE val. We assume *val* may be null if the value does not exist.

2. *The key has a S lease.* If the S lease is granted to the same session, the server returns VALUE val. Otherwise, if the value does not exist, the server returns RETRY (another session is granted with S lease and is on its process to populate the key). If the value exists, it grants the existing S lease to the session and returns VALUE val.

3. *The key has a X lease.* If the X lease granted to the same session, the server returns VALUE val. Otherwise, the server returns ABORT.

`LSet(sess_id, key, val)`. The client uses this command to set the value to the cache. If the session object exists and its state is aborted, the server deletes it and returns ABORT. Otherwise:

1. If the key has a lease that does not belong to the session, the server returns ABORT.

2. If the key has a lease belonging to the session, it is a S lease. The server puts (key, val) in the cache and returns SUCCESS.

3. If the key has a X (of another session) or no lease, the server aborts the session and returns ABORT.

`LDelete(sess_id, key)`. The client uses this command to acquire a X lease on the key (for deleting it at commit). If the key has no lease, X lease is granted to the key. If the key has S lease granted to other sessions, the server voids the S lease and aborts these sessions accordingly. It then grants a X lease to the current session. If the key has a S lease granted to the current session, it is upgraded to X lease. If the key has a X lease granted to other sessions, it also grants this lease to the current session. Finally, the server returns SUCCESS.

`LCommit(sess_id)`. The client uses this command to commit a session. If the session object exists and its state is aborted, the server deletes it and returns ABORT. Otherwise, it deletes all leases granted to the session. If the key has X lease, it is deleted from the cache. Finally, the server returns SUCCESS.

`LAbort(sess_id)`. The client uses this command to actively abort the session. If the session object exists and its state is aborted, the server deletes it and returns ABORT. Otherwise, the server releases all leases by deleting all lease objects of the keys involved in this session. The server returns SUCCESS.

5.1.1.2 Write-Through

The lease compatibility table is shown at Table 3.1. The server offers a rich set of commands to provide read-modify-write and incremental-update semantics for write-through. These include LGet with flag, LAppend, LPrepend, LIncr, LDecr and LAdd. If a session updates a key-value pair, it maintains a copy of the value (denote as *pending version*) and keeps the original version intact. When it commits, it overrides the original version with the pending version. The commands for write-through are listed below.

`LGet(sess_id, key, flag)`. The client uses this command to get the value. The flag is set to `READ_ONLY` or `RMW` if the context of the call is read-only or read-modify-write, respectively. If the flag is `READ_ONLY`, the command is processed identically as with LGet of write-around. We describe the scenario when the flag is `RMW`. If the session object exists in the cache and its state is aborted, it is deleted and the server returns ABORT. Otherwise, there may have three possibilities:

1. *The key has no lease.* The server grants X lease on the key and returns VALUE val.

2. *The key has a S lease.* The server upgrades S to X lease, causing other sessions holding the S lease to be aborted. All their leases are deleted and their session objects' statuses become abort. Finally, it returns the value (VALUE val).

3. *The key has a X lease.* If it is granted to the same session, the server returns VALUE val. If it is granted to a different session, the server cleans up this session: it deletes all the leases, pending versions and session object created by this session and returns ABORT.

`LSet(sess_id, key, val)`. The client uses this command to set the value to the cache. If the session object exists and its state is aborted, the server deletes it and returns ABORT. Otherwise:

1. If the key has a lease but it does not belong to the session, return ABORT.

2. If the session has a lease on the key, it is either a S or X lease. In both cases, the server creates a pending version (key, val) and returns SUCCESS.

3. If the key has no lease, the server grants a X lease on the key for this session, creates a pending version (key, val) and returns SUCCESS.

`LValidate(sess_id)`. The client uses this command to validate its leases prior to committing the data store transaction. If the session object exists and its state is aborted, the server deletes it and returns ABORT. Otherwise, the server checks the validity of leases hold by the session. The server may return SUCCESS when all leases are still valid, or ABORT if the server aborted the session or there is a lease expire.

`LCommit(sess_id)`. The client uses this command to commit the session. If the session object exists and its state is aborted, the server deletes it and returns ABORT. This may happen when the client issues LCommit() without a LValidate() command. Otherwise, the server releases all leases hold by this session. With each key, if its pending version exists, it is swapped with the original version. If its pending version is empty, the server deletes the original version. If its pending version does not exist, the server keeps the original version. Finally, the server returns SUCCESS.

`LAbort(sess_id)`. The client uses this command to actively abort the session. If the session object exists and its state is aborted, the server deletes it and returns ABORT. Otherwise, the server releases all leases by deleting all lease objects of the keys involved in this session. With each key, its pending version (if exists) is deleted, keeping the original version intact. The server should always return SUCCESS.

`LDelete(sess_id, key)`. The client uses this command to acquire a X lease on the key for deleting it. If the session object exists and its state is aborted, the server deletes it and returns ABORT. The server also returns ABORT if the key has a X lease granted to another session. Otherwise, there could be one of the two scenarios:

1. There is no existing lease on the key. The server grants X lease on the key, creates an empty pending version. If the original version exists, it returns SUCCESS. Otherwise, it returns NOT_FOUND;

2. If there is a S lease on the key, the cache upgrades it to a X lease, causing other sessions sharing the S lease to abort. If the pending version exists and does not empty, it replaces the pending version with an empty one and returns SUCCESS.

`LAppend(sess_id, key, delta)`. The client uses this command to append *delta*, a sequence of bytes, to the existing value (if exist). If the session object exists and its state is aborted, the server deletes it and returns ABORT. The server also returns ABORT if the key has a X lease granted to another session. Otherwise, there could be one of these scenarios:

1. There is no existing lease on the key. The server grants X lease on the key. If the value doesn't exist, it returns NOT_SUCCESS. If the value exists, it creates a pending version (by making a copy of the original version), appends delta to it and returns SUCCESS.

2. There is an existing lease on the key for the same session. If the lease is a S lease, the server upgrades it to a X lease, aborting other sessions sharing the same S lease. If the key doesn't exist, it returns NOT_SUCCESS. Otherwise, it creates a pending version, appends delta to it and returns SUCCESS. If the lease is a X lease, it checks the pending version. If the pending version is empty (meaning the key was deleted by the same session), it returns NOT_SUCCESS. If the pending version doesn't exist, it creates a pending version, appends delta to it and returns SUCCESS.

`LIncr(sess_id, key, delta)`. The description is identical to LAppend with one different: instead of appending delta to the pending version, it increments the pending version by delta[1].

As described in Chapter 3, we use commands such as multi-get, multi-append and multi-set to reduce the number of network roundtrips between the client and the cache server if the session consists of many keys. The description of these commands are as follows.

`LMultiGet(sess_id, keys, flag)`. The client uses this command to get the values of multiple keys and acquire the same lease type on them (S if flag is `READ_ONLY`, X if flag is `RMW`). It executes LGet(sess_id, key, flag) for each key. If one of the LGet commands returns ABORT, the server aborts the session and returns ABORT. Otherwise, a list of LGet replies corresponding to the keys is returned.

`LMultiSet(sess_id, keys, vals)`. The client uses this command to set multiple values to the cache. For each key, it executes `LSet(sess_id, key, val)`. If one command returns ABORT, the cache server aborts the session and returns ABORT. Otherwise, a list of LSet replies corresponding to the keys is returned.

`LMultiAppend(sess_id, keys, deltas)`. The client uses this command to append to multiple keys. For each key, it executes `LAppend(sess_id, key, delta)`. If one command returns ABORT, the cache server aborts the session and returns ABORT. Otherwise, a list of LAppend replies corresponding to the keys is returned.

---

[1] *delta* is an integer value greater than 0.

`LMultiDelete(sess_id, keys)`. The client uses this command to append to multiple keys. For each key, it executes `LDelete(sess_id, key)`. If one command returns ABORT, the cache server aborts the session and returns ABORT. Otherwise, a list of LDelete replies corresponding to the keys is returned.

5.1.1.3 Write-back Policy

Applications may generate buffered writes and store them as key-value pairs in the cache. These buffered writes must be pinned in memory for the cache eviction policy to bypass them. The set of commands for write-back policy is identical with write-through except one change: an additional flag `is_pinned` for `LSet` and `LAdd` to specify whether the key-value pair is pinned in the cache. When this flag is set, the server sets this flag on the pending version of the key. When the session commits, if the pending version has this flag set, the server overrides the original version with the pending version and pins it in memory.

If a key-value pair is pinned, following commands that modify its value also keep this key-value pair pinned. For example, a `LAppend` or `LIncr` on a pinned key-value pair keeps it pinned in memory. A pinned key-value pair is deleted only when a client issues a session having `LDelete` request to delete it. The server does not delete pinned key-value pairs, even when their leases expire.

When a CMI with buffered writes runs out of memory, it may not have space for coming key-value pairs because it cannot evict any cache entry, assuming they are all pinned. In such scenario, it returns NOT_STORED to the client. The client may choose to either retry or abort the session.

The `is_pinned` flag only guarantees that buffered writes are not evicted by a cache policy. To provide durability for buffered writes, they must be replicated to

multiple cache servers, store on non-volatile memory layer such as NVDIMM-N, or asynchronously write to a persistent data store such as BerkeleyDB, see Chapter 6.

5.1.2 Client

We may extend an existing client such as WhalinClient [74] to implement methods that support new commands. Each application thread maintains an instance of the client (denoted as *mc*) to manipulate sessions generated by the thread. We describe the provided methods as follows.

`mc.GetActiveSessionId()`. This method is transparent to the developer. It is invoked by other methods (described below) asking for a session id. If there is an active session id, it is returned. Otherwise, it first checks to see if a session id can be provided by the current range the client holds. If the range is exhausted, it requests the coordinator for a new range. Finally, a session id is identified. It becomes the active session id and is returned to the caller.

`mc.StartSession(sess_id)`. An application may use this method to start a session and explicitly provide the session id. This is helpful if the session id carries additional information. NgCache uses this command because its generated session id carries the sesion queue number.

`mc.LGet(key, flag)`. It invokes GetActiveSessionId() to get the session id. The flag is either `READ_ONLY` or `RMW`. The session id is included in the request to the cache server. If the cache server returns with a value (either null or not null), it is returned to the caller. Otherwise, if the cache server returns with RETRY, the client sleeps for a few milliseconds before retry. If the cache server returns with abort, it throws SessionAbortException to the caller.

`mc.LSet(key, val, is_pinned)`. It invokes GetActiveSessionId() to get the session id. The session id is included in the request to the cache server. The server

may return with SUCCESS, NOT_SUCCESS or ABORT. It returns true, false or throws SessionAbortException to the caller, respectively.

`mc.LGet(key)` and `mc.LSet(key, val)` are other variants of LGet and LSet. `mc.LGet(key)` is identical to `mc.LGet(key, flag)` with the flag set to `READ_ONLY`. It is used with the write-around policy. `mc.LSet(key, val)` is identical to `mc.LSet(key, val, is_pinned)` with `is_pinned` is *false*. It is used with the write-though policy.

Commands `mc.LDelete(key)`, `mc.LValidate()`, `mc.LAdd(key, val)`, `mc.LAppend(key, val)`, `mc.LPrepend(key, val)`, `mc.LIncr(key, val)`, `mc.LDecr(key, val)` are handled similar to `mc.LSet(key, val)`.

`mc.LCommit()`. It invokes GetActiveSessionId() to get the session id. The session id is included in the corresponding request to the cache server. The server returns with either SUCCESS or ABORT. It returns true or throws SessionAbortException to the caller, respectively.

`mc.LAbort()`. It invokes GetActiveSessionId() to get the session id. The session id is included in the corresponding request to the cache server. The server always returns with SUCCESS. It returns true to the caller.

The following methods are implemented to support server commands on multiple keys.

`mc.LMultiGet(keys, vals, flag)`. It invokes GetActiveSessionId() to get the session id. The session id is included in the corresponding request to the cache server. The server may return either 1) a list of values (either null or not null), 2) ABORT or 3) a list of values (either null or not null) and RETRY for some provided keys. With the first, it returns the values to the caller. With the second, it throws SessionAbortException to the caller. With the third, it sleeps a few

milliseconds and issues LMultiGet for the keys corresponding to the RETRYs. It repeatedly does so until the server replies with either 1) or 2).

`mc.LMultiSet(keys, vals, is_pinned)`. It invokes GetActiveSessionId() to get the session id. The session id is included in the corresponding request to the cache server. The server may return either SUCCESS or ABORT. It returns true or false to the caller, respectively.

`mc.LMultiDelete(keys)`. It invokes GetActiveSessionId() to get the session id. The session id is included in the corresponding request to the cache server. The server may return either SUCCESS or ABORT. It returns true or false to the caller, respectively.

`mc.LMultiAppend(keys, deltas)`. It invokes GetActiveSessionId() to get the session id. The session id is included in the corresponding request to the cache server. The server may return either SUCCESS or ABORT. It returns true or false to the caller, respectively.

5.1.3 Handle Multiple CMIs

With a deployment consisting of multiple CMIs, a session may have keys belonging to different CMIs. The cache client gets a list of the CMIs from the coordinator. The application is required to implement a hashing function that takes the input is the key and the output is the identify of the CMI that stores it.

With a command consisting of a single key, the hash function is invoked to identify the CMI. The client then directs the request to it. With a command consisting of multiple keys, we have a mapping between CMIs and keys. The command are splitted to multiple sub-commands where each has keys belonging to one CMI grouped together. Each sub-command is directed to its targeted CMI

to be processed in parallel. Finally, the aggregated results are returned to the application.

One CMI returning abort causes the session to abort. Once the session validates successfully to one CMI, the session may still abort if a) another CMI returns abort to the client, or b) the client fails (CMI aborts the session when the lease expires).

It may happen that the session commits successfully at some CMIs and the client crashes before sending the commit messages to others. When the leases expire, they abort the session, deleting any key-value pairs involved in this session.

5.2 NgCache

Our evaluations at Chapter 3 show that write-back improves performance significantly. However, its complex designs may challenge developers with the implementation. Developers must write code to handle these complexities, and some of them are burdensome and error prone that cost time and money to debug and maintain. The challenges are:

- Extend the existing client logic to implement session.
- Store cache entries properly, including deciding how to partition them to cache servers, be aware of the fact that a key-value pair may not be stored or evicted.
- Provide durability, including deciding which cache entries are pinned, how to replicate buffered writes to multiple CMIs.

NgCache is a general framework that supports developers to quickly implement a cache policy for the CADS architecture. The cache policy may be either write-around, write-through or write-back. It hides the burden of implementing logical

executions of reads and writes to maintain the data store and the cache consistent. It manages the cache leases, buffered writes durability by pinning and replicating them in multiple cache servers transparently. The framework is realized by a library used at the application servers. NgCache provides the following benefits:

- Guide the developers with interfaces to implement a write policy. Some interfaces are mandatory, while others are optional.
- Hide the cache operations from developers. From a developer's viewpoint, a session may be either commits or aborts, and he only needs to handle each case properly (retry the session, display error message,...). NgCache, in this sense, is very similar to a database client library such as JDBC or MongoDBClient.
- Pin a buffered write in memory, and replicates it to R CMIs to provide durability transparently.[2]
- Support either a SQL or NoSQL data store. For the system to provide consistency, it is required that the data store must support transactions and is configured to provide consistency at the same level.
- By default, it uses LeCaF framework (Section 5.1) in its implementation. NgCache provides an interfaces for cache commands so that developers may implement their own cache framework. However, note that without a lease framework as detailed in Section 5.1, handling consistency becomes the responsibility of developers.

A required configuration file specifies multiple parameter values. The parameters include the cache servers, the cache policy, the connection string to the data

[2]R is a configurable parameter

store and the number of background worker threads (if the cache policy is write-back).

Each application thread initializes a NgCache object by invoking its initialize method Initialize(cacheStore, writeBack) where cacheStore and writeBack are instances of implementations of CacheStore and WriteBack, respectively. While CacheStore provides interfaces to support write-around and write-through policies, WriteBack provides interfaces to support write-back policy. Developers must provide implementations for these interfaces to enable the desired cache policy accordingly, see Tables 5.2 and 5.3.

Algorithm 8: Implementing a session with NgCache.

```
1  ngCache.startSession(prefix);
2  try:
3      foreach data store statement S_i do
4          if S_i is a read statement then
5              queryResult ← ngCache.readStatement(S_i);
6          if S_i is a write statement then
7              ngCache.writeStatement(S_i);
8      res ← ngCache.preCommitSession();
9      if res is success then
10         Commit data store transaction;
11         ngCache.commitSession();
12     else
13         Rollback data store transaction;
14         ngCache.abortSession();
15 catch SessionAbortException:
16     Rollback data store transaction;
17     ngCache.abortSession();
```

NgCache provides methods to facilitate sessions. A session implemented by a developer starts by invoking the method startSession() on the NgCache object. It then invokes readStatement(query) or writeStatement(DML) to execute read or

write statements. Prior to committing data store transaction, it validates the session with the cache. If the validation fails, it rollbacks the data store transaction and aborts by invoking abortSession(). Otherwise, if the validation succeeds, it commits the data store transaction and commits the session by invoking commitSession(). A session may be aborted by the cache server at any time during the session life time. If this happens, ngCache methods throw SessionAbortException. The session then rollbacks the data store transaction and aborts the session by invoking abortSession(), see Algorithm 8. We discuss each method of NgCache in turn.

ngCache.startSession(prefix). The client uses this method to start a new session. A unique session id is generated. *prefix* is an optional parameter that is included with the session id. It is used as a factor for deciding which session queue the session id belongs to (for example, with TPC-C, *prefix* is a warehouse id). The generated session id is kept privately by the ngCache object and used during the session lifetime. It is set directly to LeCaF, bypassing it generating a session id by itself. Session id is deleted when the session either commits or aborts when the application invokes commitSession() or abortSession().

Two methods **ngCache.readStatement(query)** and **ngCache.writeStatement(DML)** may invoke calls to the interfaces of CacheStore and WriteBack implemented by the developer. It is the responsibility of him to provide the mapping between select and DML statements to the cache entries and to the buffered writes (with write-back policy). Interfaces in CacheStore must be implemented to support write-around or write-through policies, while interfaces in both CacheStore and WriteBack must be implemented to support write-back policy.

Table 5.2 lists the interfaces in CacheStore. *result* and *entries* are instances of classes CacheEntry and QueryResult, respectively. They are base classes that can easily be extended to fit with the need of applications.

Interface	Description
`getReferencedKeysFromQuery(query)`	Given the query of a read, return a set of the referenced keys. This command is required to support write-around, write-through or write-back.
`getImpactedKeysFromDml(dml)`	Given a DML, return the set of impacted keys. This command is required to support write-around, write-through or write-back.
`dmlDataStore(dml) throws Exception`	Given a DML, execute it to the data store. This command is required to support write-around, write-through or write-back.
`modifyCacheEntries(dml, keys)`	Given a DML and its impacted keys, return a dictionary of the changes that must be applied to the values of those keys. This command is required to support write-through or write-back.
`computeCacheEntries(query, result)`	Given the query result, compute the cache entries to be inserted in the cache. This command is required to support write-around, write-through or write-back.
`computeQueryResult(query, entries)`	Given the set of cache entries, convert it back to the query result. This command is required to support write-around, write-through or write-back.

Table 5.2: CacheStore interfaces.

Table 5.3 lists the interfaces in WriteBack. These interfaces are specifically used to support write-back policy. Hence, an implementation for write-around or write-through is not required to implement them.

Algorithm 9 and 10 describe how `readStatement(query)` and `writeStatement(dml)` are implemented internally. It illustrates how the interfaces implemented by developers are referenced by the library.

Interface	Description
`rationalizeRead(query)`	Given a query, return the list of data item identifiers this query references.
`rationalizeWrite(dml)`	Given a dml, return the list of changes that must be buffered in the cache with write-back policy.
`applySessions(sessions, conn) throws Exception`	Apply the provided list of sessions as one transaction to the data store.
`applyBufferedWriteToQueryResult( queryResult, bufferedWrite)`	Apply the buffered write to the query result.

Table 5.3: WriteBack interfaces.

readStatement(query). The pseudo-code is shown at Algorithm 9. It invokes `getReferencedKeysFromQuery()` interface of CacheStore to get a list of referenced keys for the query (Step 1). Next, it issues a multi-get command to the cache. If all values are found, it computes the query result and return (Steps 2-5). Otherwise, if there is at least one cache miss, it queries the data store for a query result (Step 6). If the cache policy is write-back, it uses `rationalizeRead()` interface to identify the mappings corresponding to the query. With a list of session ids found from these mappings, it gets their buffered writes from the cache. The buffered writes are applied to the query result by invoking the WriteBack interface `applyBufferedWriteToQueryResult()` to produce the latest result (Steps 7-12). Finally, it computes the missing key-value pairs and uses `LMultiSet()` command to put them in the cache (Steps 13-16).

writeStatement(DML). The pseudo-code is shown at Algorithm 10. It invokes `getImpactedKeysFromDml()` interface of CacheStore with the given DML to get a list of impacted keys (Step 1). If the cache policy is write-around, the keys are added to a local list (Steps 2-3). If the cache policy is write-though or write-back, it invokes `modifyCacheEntries()` interface of CacheStore to get a list

Algorithm 9: ngCache.readStatement(query)

```
   // Check the cache first
1  keys ← cacheStore.getReferencedKeysFromQuery(query);
2  vals ← mc.LMultiGet((keys, flag=READ_ONLY);
3  if all cache hits then
4      queryResult ← cacheStore.computeQueryResult(query, vals);
5      return queryResult;
   // There is at least one cache miss
6  queryResult ← Query result from the data store;
7  if cache policy is write-back then
8      mappingKeys ← writeBack.rationalizeRead(query);
9      sessionIds ← getSessionIdsFromMappings(mappingKeys);
10     foreach sessionId in sessionIds do
11         bufferedWrite ← getBufferedWrite(sessionId);
12         writeBack.applyBufferedWriteToQueryResult(queryResult,
             bufferedWrite);
   // Compute cache entries to put in the cache
13 cacheEntries ← cacheStore.computeCacheEntries(query, queryResult);
14 missKeys, missVals ← Prepare the missing keys and values;
15 mc.LMultiSet(missKeys, missVals);
16 return queryResult;
```

of deltas (Steps 4-6). These deltas are used to update the application regular key-value pairs. If the cache policy is write-around or write-through, it applies the DML to the data store synchronously (Steps 7-8). Otherwise, if the cache policy is write-back, it invokes `rationalizeWrite()` to get a list of mapping keys and new changes (Steps 9-11). Finally, it returns SUCCESS. As shown, this method only generates changes and stores them locally (*newDeltas*, *mappingKeys*, *newChanges*). These changes are applied to the cache by the ngCache.validate() method.

ngCache.preCommitSession(). This method is called prior to committing the data store transaction. It uses LeCaF methods such as LMultiGet, LMultiSet, LMultiDelete and LMultiAppend to 1) delete the application cache entries using *deletedKeys* with write-around, or update them with write-through and write-back

Algorithm 10: ngCache.writeStatement(DML)

```
1  keys ← cacheStore.getImpactedKeysFromDml(DML);
2  if cache policy is write-around then
3      Store keys to a local list deletedKeys to be deleted;
4  if cache policy is write-through or write-back then
5      newDeltas ← cacheStore.modifyCacheEntries(DML, keys);
6      Store newDeltas locally;
   // Apply the write to the data store synchronously if the cache policy is
      either write-around or write-through
7  if cache policy is write-around or write-through then
8      cacheStore.dmlDataStore(dml);
9  if cache policy is write-back then
10     mappingKeys, newChanges ← cacheStore.rationalizeWrite(DML);
11     Store mappingKeys and newChanges locally;
12 return success
```

using *newDeltas*, and 2) store and pin the buffered write using *newChanges*, append the session id to the mapping keys using *mappingKeys*, and append the session id to the session queue with write-back. Finally, it invokes LeCaF LValidate() method to ensure all leases granted to this session are valid. If one of the cache operations throws SessionAbortException, it throws SessionAbortException. Otherwise, it returns true (success), seee Algorithm 11.

ngCache.commitSession(). The session proceeds to commit by invoking LeCaF LCommit() method.

Algorithm 11: ngCache.preCommitSession()

```
1  if cache policy is write-around then
       // Delete the impacted keys
2      mc.LMultiDelete(deletedKeys);
3  if cache policy is write-back then
       // Store the buffered write in the cache
4      buffVal ← Construct the buffered write value with newChanges;
5      mc.LSet(sess_id, buffVal); // Store with the key sess_id
       // Append sess_id to the mapping keys and the queue
6      queueKey ← Identify the queue key from the session id;
7      keys ← mappingKeys ∪ { queueKey };
8      mc.LMultiAppend(keys, sess_id);
9  if cache policy is write-through or write-back then
       // Update the impacted key-value pairs
       // For simplicity, we illustrate with read-modify-write semantic
10     keys ← Get the cached keys from newDeltas;
11     vals ← mc.LMultiGet(keys, flag=RMW);
12     newVals ← {};
13     for key in keys do
14         newVal ← Update val with the corresponding delta from newDeltas;
15         Add newVal to newVals;
16     mc.LMultiSet(keys, newVals);
17     mc.LValidate();
```

Chapter 6

Future Plan

6.1 Persistent Cache

Replicating buffered writes to multiple CMIs on different cache servers is not sufficient to provide durability because a data center failure may cause all replicas to fail. To enhance durability, buffered writes may be stored asynchronously to an embedded persistent data store such as BerkeleyDB [59]. A persistent data store may also increase cache hit rate when the memory is limited. Regular cache entries may be evicted to the data store. Once they are referenced again, the cache brings them back to its memory. A design for persistent cache must ensure consistency and minimize the impact of writing buffered writes to the persistent data store to the cache foreground thread. The performance of write-back with three replicas must outperform a solution using only the data store for this approach to provide benefits.

6.2 Max Pinned Memory Size

With limited cache, there is not sufficient memory for storing both cache entries and key-value pairs (buffered writes, mappings and Queues) that are necessary to enable the write-back policy. They are pinned and cannot be evicted. Fixing a large value for max pinned memory size leaves small cache size for cache entries. The coming key-value pairs compete for this portion, lowering the cache hit ratio.

On the other hand, a low value for max pinned memory size increases the number of writes that must switch to write-through because they fail to incorporate their changes in the cache. Intelligent algorithms are required to adjust the pinned memory size to maximize the system performance.

6.3 Write-through vs. Write-back

As shown in Section 3.3, write-back may not provide any benefits, or even worse when comparing with write-through with limited cache. Write-back must switch to write-through with the overhead of trying to buffer writes in the cache to fail. Intelligent algorithms are required to switching the cache policy to maximize its performance.

6.4 Number of Background Worker Threads

With abundant cache memory, the system performance is impacted by the background worker threads when they apply the writes to the data store, as shown in Section 3.3.1.1. Dynamically adjusting the number of background threads to balance the system performance and the percentage of writes applied to the data store is an interesting research direction.

6.5 Prioritizing Buffered Writes

Skew access patterns cause some data items to be updated more frequently than others. Prioritizing others before applying these buffered writes allow more changes to be buffered in these buffered writes. These changes are then merged and applied to the data store efficiently.

6.6 Transparent Cache

The design of interfaces for the cache framework at Section 5.2 requires a developer to identify the cache entries given a statement (read or write). He also needs to provide a mapping between a read/write statement to data items that identify their buffered writes. A transparent caching framework hides all of these complexities from developers. The challenge is to identify the cache entries referenced by a read statement or impacted by a write statement. This requires the library to extract the query or dml to get attributes and their values, decide whether a read statement should or should not be cached, or a transaction should be executed with write-back policy. This adds complexity to the library, but simplifies the interfaces for the cache framework a lot and makes it much easier and useful for developers.

A cache such as TxCache [62] supports transparent caching to a certain extent. A developer may specify to cache or not cache a query. Based on the object relational mapping (ORM), it constructs the cache entry and identifies the writes impacting it. It does not handle multi-statement transactions. Moreover, it does not support write-back policy and does not have the concept of buffered writes.

Reference List

[1] Redis Persistence, https://redis.io/topics/persistence.

[2] Redis Replication, https://redis.io/topics/sentinel.

[3] repcached: Add Data Replication Feature to Memcached 1.2.x, http://repcached.lab.klab.org/.

[4] Y. Alabdulkarim, M. Almaymoni, Z. Cao, S. Ghandeharizadeh, H. Nguyen, and L. Song. A Comparison of Flashcache with IQ-Twemcached. In *IEEE CloudDM*, 2016.

[5] Y. Alabdulkarim, M. Almaymoni, Z. Cao, S. Ghandeharizadeh, H. Nguyen, and L. Song. A comparison of Flashcache with IQ-Twemcached. In *ICDE Workshops*, 2016.

[6] Amazon DynamoDB. https://aws.amazon.com/dynamodb/ and https://aws.amazon.com/dynamodb/pricing/, 2016.

[7] S. S. (antirez) and M. Kleppmann. Redlease and How To do distributed locking. http://redis.io/topics/distlock and http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html.

[8] Apache. Ignite - In-Memory Data Fabric, https://ignite.apache.org/, 2016.

[9] S. Apart. Memcached Specification, http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt.

[10] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[12] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Quantifying eventual consistency with PBS. *Commun. ACM*, 57(8), 2014.

[13] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR*, January 2013.

[14] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *USENIX ATC*, 2017.

[15] E. Brewer. Towards Robust Distributed Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2000.

[16] N. Bronson, T. Lento, and J. L. Wiener. Open Data Challenges at Facebook. In *ICDE*, 2015.

[17] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side Flash Caching for the Data Center. In *MSST*, 2012.

[18] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *NSDI*, pages 379–392, 2016.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.

[20] S. Daniel and S. Jafri. Using NetApp Flash Cache (PAM II) in Online Transaction Processing. *NetApp White Paper*, 2009.

[21] 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 *SOSP*, 2007.

[22] DELL. Dell Fluid Cache for Storage Area Networks, http://www.dell.com/learn/us/en/04/solutions/fluid-cache-san, 2014.

[23] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4):277–288, 2013.

[24] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, 2013.

[25] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124), 2004.

[26] Flux Research Group. Emulab, https://www.emulab.net/.

[27] S. Ghandeharizadeh and et. al. A Demonstration of KOSAR: An Elastic, Scalable, Highly Available SQL Middleware. In *ACM Middleware*, 2014.

[28] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores. *Middleware*, 2014.

[29] S. Ghandeharizadeh and J. Yap. Cache Augmented Database Management Systems. In *ACM SIGMOD DBSocial Workshop*, June 2013.

[30] S. Ghandeharizadeh, J. Yap, and H. Nguyen. IQ-Twemcached. http://dblab.usc.edu/users/IQ/.

[31] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. *Middleware*, December 2014.

[32] Google. Guava: Core Libraries for Java, https://github.com/google/guava, 2015.

[33] Google App Engine Pricing, Apps are free within a usage limit that is reset daily. https://cloud.google.com/appengine/, 2016.

[34] G. Graefe. The Five-Minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *DaMoN*, page 6, 2007.

[35] C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[36] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, pages 677–680. Morgan Kaufmann, 1993.

[37] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[38] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash Caching on the Storage Client. In *USENIXATC*, 2013.

[39] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer. Flash Caching on the Storage Client. In *USENIX ATC'13 Proceedings of the 2013 USENIX conference on Annual Technical Conference*. USENIX Association, 2013.

[40] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, July 2015.

[41] IBM. WebSphere eXtreme Scale, https://goo.gl/smgC3W.

[42] Infinispan. Infinispan, http://infinispan.org/.

[43] S. Irani, J. Lam, and S. Ghandeharizadeh. Cache Replacement with Memory Allocation. *ALENEX*, 2015.

[44] H. Kim, I. Koltsidas, N. Ioannou, S. Seshadri, P. Muench, C. Dickey, and L. Chiu. Flash-Conscious Cache Population for Enterprise Database Workloads. In *Fifth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2014.

[45] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[46] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *POPL*, 2016.

[47] D. Liu, N. Mi, J. Tai, X. Zhu, and J. Lo. VFRM: Flash Resource Manager in VMWare ESX Server. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–7. IEEE, 2014.

[48] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP*, 2011.

[49] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual consistency. *Commun. ACM*, 57(5):61–68, May 2014.

[50] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[51] N. Lynch and S. Gilbert. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT New*, 33:51–59, 2002.

[52] D. Mituzas. Flashcache at Facebook: From 2010 to 2013 and Beyond, https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920, 2010.

[53] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[54] Mongo Inc. Mongo Atomicity and Transactions, https://docs.mongodb.com/manual/core/write-operations-atomicity/.

[55] MongoDB Inc. MongoDB, https://www.mongodb.com/.

[56] MySQL. Designing and Implementing Scalable Applications with Memcached and MySQL, A MySQL White Paper, June 2008, http://www.mysql.com/why-mysql/memcached/.

[57] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 15–28, Berkeley, CA, USA, 2008. USENIX Association.

[58] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, pages 385–398, Berkeley, CA, 2013. USENIX.

[59] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[60] Oracle. Oracle Coherence, http://www.oracle.com/technetwork/middleware/coherence.

[61] E. Pitoura and B. Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, 1995.

[62] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in An Application Data Cache. 2010.

[63] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI*. USENIX, October 2010.

[64] redis. Redis, https://redis.io/.

[65] Y. A. S. Ghandeharizadeh and H. Nguyen. CPR: Client-Side Processing of Range Predicates, USC Database Laboratory Technical Report Number 2018-03, http://dblab.usc.edu/Users/papers/CPR.pdf.

[66] A. Sainio. NVDIMM: Changes are Here So WhatâĂŹs Next. *In-Memory Computing Summit*, 2016.

[67] W. Stearns and K. Overstreet. Bcache: Caching Beyond Just RAM. https://lwn.net/Articles/394672/,http://bcache.evilpiepirate.org/, 2010.

[68] STEC. EnhanceIO SSD Caching Software, https://github.com/stec-inc/EnhanceIO, 2012.

[69] Terracotta. Ehcache, http://ehcache.org/documentation/overview.html.

[70] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.

[71] Transaction Processiong Performance Council. TPC-C, http://www.tpc.org/tpcc/.

[72] P. Viotti and M. Vukolić. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.*, 49(1), June 2016.

[73] W. Vogels. Eventually Consistent. *Communications of the ACM, Vol. 52, No. 1*, pages 40–45, January 2009.

[74] G. Whalin, X. Wang, and M. Li. Whalin memcached Client Version 2.6.1, http://github.com/gwhalin/Memcached-Java-Client/releases/tag/release_2.6.1.