Algorithmic Engineering Towards More Efficient Key-Value Systems

Bin Fan

CMU-CS-13-126
December 18, 2013

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, Chair
Michael Kaminsky, Intel Labs
Garth A. Gibson
Edmund B. Nightingale, Microsoft Research

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Memory Efficiency, Hash Table, Bloom Filter, Caching, Load Balancing

Dedicated to my family

Abstract

Distributed key-value systems have been widely used as elemental components of many Internet-scale services at sites such as Amazon, Facebook and Twitter. This thesis examines a system design approach to scale existing key-value systems, both horizontally and vertically, by carefully engineering and integrating techniques that are grounded in recent theory but also informed by underlying architectures and expected workloads in practice. As a case study, we re-design FAWN-KV—a distributed key-value cluster consisting of "wimpy" key-value nodes—to use less memory but achieve higher throughput even in the worst case.

First, to improve the worst-case throughput of a FAWN-KV system, we propose a randomized load balancing scheme that can fully utilize all the nodes regardless of their query distribution. We analytically prove and empirically demonstrate that deploying a very small but extremely fast load balancer at FAWN-KV can effectively prevent uneven or dynamic workloads creating hotspots on individual nodes. Moreover, our analysis provides service designers a mathematically tractable approach to estimate the worst-case throughput and also avoid drastic over-provisioning in similar distributed key-value systems.

Second, to implement the high-speed load balancer and also to improve the space efficiency of individual key-value nodes, we propose novel data structures and algorithms, including the *cuckoo filter*, a Bloom filter replacement that is high-speed, highly compact and delete-supporting, and *optimistic cuckoo hashing*, a fast and space-efficient hashing scheme that scales on multiple CPUs. Both algorithms are built upon conventional cuckoo hashing but are optimized for our target architectures and workloads. Using them as building blocks, we design and implement MemC3 to serve transient data from DRAM with high throughput and low-latency retrievals, and SILT to provide cost-effective access to persistent data on flash storage with extremely small memory footprint (e.g., 0.7 bytes per entry).

Acknowledgments

First and foremost, I am greatly indebted to my advisor, Dave Andersen. I am so fortunate to have Dave as my advisor to learn so many things. When I first arrived at CMU, I had almost no background in system building. Through the years Dave guided me, inspired my ideas, encouraged my progress and steered my research. Dave taught me skills from configuring `.bashrc` to using various fancy compiler/hardware tricks for performance; he motivated and helped me with his great vision and also his persistence in solving hard but interesting problems; he reviewed my source code and often found bugs; he corrected every grammar mistake I made in my writing, and taught me the precise and concise way to write technical content; he polished every presentation of mine from the slides outline to the figure font. For these, and too much more, I express my utmost gratitude to him. Without Dave, I will never reach anywhere even close to my current state.

I also want to specially thank Michael Kaminsky, who has been collaborating with me for such a long time on every project of mine in CMU. For too many times, my research benefited from the inspiring discussions with Michael. He sets me a wonderful example of organizing everything well and making steady and substantial progress even with multiple concurrent projects going-on. Both Dave and Micheal have been central in each stage of this thesis.

Garth Gibson and Ed Nightingale were kind enough to serve on my dissertation committee. I started working with Garth from the DiskReduce project. At that time, I grabbed so many cycles from Garth that new students thought Garth was my advisor. I was always amazed by the depth and breadth of Garth's knowledge in storage and distributed systems, as well as his great vision in identifying the most interesting and challenging part in research. In 2011, I spent a summer with Ed in building 99 of Microsoft Research Redmond as a student intern. It was my best summer, after years the system we built together still makes me exciting. More importantly, I learned from Ed that how many creative ideas can be realized when a brilliant mind is hard-working, and thinking differently.

My work as a Ph.D. student was made possible, largely due to the collaboration with Hyeontaek Lim. Without hacking with Hyeontaek on SILT [59] and small-cache [40], I would never be able to appreciate the art of system building. I also want to thank a group of PDL/CMCL friends: Wittawat Tantisiriroj, Lin Xiao, Kai Ren, Iulian Moraru, Dong Zhou, Amar Phanishayee, Vijay Vasudevan, Dongsu Han and Swapnil Patil. I am so lucky to have you around to help me solve various random problems in hacking or writing, teach me all kinds of useful skills, share with me the interesting news, and update me the popular restaurants.

I also owe a great deal of gratitude to John Lui and Dah-Ming Chiu, who co-advised me at the Chinese University of Hong Kong. I would not have been in CMU without their guidance and help. It was always a thrill for me to hear John and Dah-Ming's vision and understanding in networking systems.

My study at CMU has been helped and improved under the help of many other faculty members: Greg Ganger, Peter Steenkiste, Andy Pavlo, Srinivasan Seshan, Hui Zhang. Thank you so much for the thought-provoking questions on PDL visiting days/retreats and Tuesday seminars (and the Pizza). I learned how to evaluate my research from different perspectives.

Furthermore, I thank these warm-hearted people who were frequently bothered by me but always effective to support me: Deborah Cavlovich, Angela Miller, Karen Lindenfelser, Joan Digney, and Kathy McNiff. I can't thank you enough for how much you have done.

Finally, this thesis would not be possible without the support from my parents Zhibing Fan and Pingping Wen, and of course my wife, Shuang Su. Their constant support as well as the serious efforts in forgiving my unbalanced work/life is my driving force. My last gratefulness goes to my friends in Pittsburgh. I am so fortunate to have years spent with such a group of incredibly bright and fun people: Bin Fu, Fan Guo, Guang Xiang, Jialiu Lin, Junchen Jiang, Junming Yin, Long Qin, Nan Li, Pingzhong Tang, Xi Liu, Xiong Zhang, Yanan Chen, Yin Zhang, Yuandong Tian, Yunchuan Kong, Zongwei Zhou. You are forever my best friends and my memory at CMU becomes full of happiness only with you.

Declaration of Collaboration This thesis has benefited from many collaborators, specifically

- David Andersen and Michael Kaminsky are contributors to all the work included in this thesis.

- Hyeontaek Lim contributed to this thesis in multiple joint papers [40] [59]. He made a great amount of contribution in our experiments in Chapter 2. He also designed and implemented the multi-store architecture of SILT in Chapter 5. Especially, he proposed and implemented the entropy-encoded trie that is used in SILT as an extremely memory-efficient key-value index.

- Michael Mitzenmacher and Rasmus Pagh contributed to the theoretic analysis of partial-key cuckoo hashing, which is a core technique used in Chapter 3 and Chapter 4.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data intensive computing keeps growing rapidly in importance and scale [24, 33, 36]. Today, storage clusters with thousands of nodes and beyond have become commonplace in many Internet companies. Familiar examples include Google's BigTable and GFS cells (1000 to 7000 nodes in one cluster [43]), Facebook's photo storage (20 petabytes of data [14]), Microsoft's data mining cluster (1800 nodes [52]), and Yahoo's Hammer cluster (3800 nodes [76]). These cluster systems scale their performance and ensure load balance across all nodes by using combinations of techniques including:

- partitioning: spreading the entire data-set across a larger number of nodes, where each node handles a different subset of the requirements. In particular, consistent hashing schemes that map data items to nodes by hashing as used in Chord [54, 87] are widely used due to their simplicity and ability to support incremental growth. Many systems that implement consistent hashing also use "virtual nodes" to improve the quality of load balancing, where each physical server acts as several different nodes in the consistent hashing ring [30].

- replication: storing each piece of data multiple copies on different nodes. For example, Google file system [45] and Hadoop file system [2] both replicate each data chunk several times in the cluster, with the minimum being three to ensure availability against disk failures or to meet high demand of accesses.

However as the service complexity continues growing, it becomes increasingly challenging to ensure load balancing. For example, Facebook deployed a large pool of Memcached servers to serve terabytes of user data from DRAM without reading disks [73]. But because the popularity of their queries is highly skewed, they have to partition the data very carefully to prevent those popular data items from overloading servers. As another example, Google reported that a single web search query usually consults tens to hundreds of different backend servers, and as a result the total response time of a query from the cluster is determined by the response time of the slowest node [32]. Hence, when even a small fraction of servers are overloaded, their slow response time could substantially degrade the service. Amazon's Dynamo [36] uses consistent hashing, virtual nodes, and replication. However, its authors report that 10% of nodes have at least 15% higher load than the average load almost all of the time. These techniques are usually good at balancing the *static* component of load—the constant storage or memory capacity required on individual nodes, which we typically refer to as the amount of data they store—while being not as good at handling the *dynamic* load of handling queries as they arrive.

The first goal of this thesis is to allow several types of back-end cloud services to meet service-level objectives (SLOs) without drastic over-provisioning, regardless of the possibly changing query distribution. In this thesis, we define SLOs as particular rates of queries a cluster system promises to complete to its customers, at a very high probability (e.g., complete 10 million queries per second 99.99% of the service time). It is important to ensure high SLOs because these storage clusters are often used as building blocks for other systems and applications. Thus, higher SLOs greatly benefit these systems and applications in simpler design and more reliable implementation.

Improving the load balance is one way to efficiently utilize the aggregated capacity of a storage cluster and scale out its performance; it is equally important to improve the resource efficiency of each individual node. As the cluster size keeps growing, cost concerns dictate the efforts to scale up the capacity of each node. For example, in datacenters, memory space in particular is increasingly expensive and scarce relative to external storage capacity such as hard disks or SSDs. For example, the cost of Amazon EC2 instances is currently dominated by DRAM rather than CPU capacity [7]. Intensifying this issue, the growth of DRAM capacity has been much slower than the growth of disk or flash storage capacity [59], requiring storage servers to index increasingly larger data sets by using more space-efficient in-memory indexes (i.e., using fewer bits in DRAM for one data entry on disk/SSD). Recent proposals have started examining memory-efficient indexing schemes for building high-performance data stores [8, 10, 13, 34, 35, 72, 89], but these solutions either compromise performance [35, 72, 89] or still impose a significant amount of overhead in space [8, 9, 13, 34]. As a result, there has been an increasing demand for more memory-efficient index schemes to *scale up* the capacity of individual storage nodes.

Given the above observations, we therefore apply in this dissertation a system design approach that carefully applies techniques that are both grounded in recent theory (e.g., cuckoo hashing [81] and randomized load balancing) and informed by the underlying hardware and expected workloads, to design more efficient key-value systems. As a case study, we build a distributed key-value cluster based on FAWN-KV [10]. The goal is to achieve *higher throughput* with *lower memory overhead* on each node, and maintain *high overall throughput under different or even adversarial workloads.* To this end, this thesis tackles this problem in two aspects:

- *For certain types of many-node key-value systems, we present a load-balancing and caching mechanism for fully utilizing the capacity of all nodes under arbitrary workloads, even adversarial ones.*

  This thesis analytically proves and empirically demonstrates that a combination of caching and randomized load balancing is an effective approach to prevent uneven or dynamic workloads from creating hotspots on individual nodes. Particularly, we present a mathematically tractable approach to analyze this dynamic load balancing scheme and better estimate its achievable service-level objectives in the worst case. Our result is useful for cloud service providers to avoid drastic over-provisioning in capacity planning.

- *For each individual key-value node, we explore novel algorithms and data structures to improve the per-node efficiency.*

  In this thesis, we introduce the *cuckoo filter*, a Bloom filter replacement that is fast, compact, and supports deletion; and *optimistic cuckoo hashing*, a space-efficient concurrent hashing scheme. Using them as building blocks, we design and implement two state-of-the-art key-value stores: MemC3 serves transient data from DRAM with high throughput and low latency, while SILT provides cost-effective access to persistent storage on flash devices with extremely small memory footprint (e.g., 0.7 bytes per entry).

This dissertation makes the following contributions:

- The analysis of provable randomized load balancing in a distributed key-value cluster, which allows cloud service providers to meet service-level objectives (SLOs) for handling a particular rate of queries for the worst-case workloads, without the need for drastic over-provisioning;

- The design and implementation of a fast, concurrent and space-efficient hash table based on cuckoo hashing [81], where parallelizing cuckoo hashing was considered an open problem [68] and to our knowledge, our scheme is the first to support single-write/multi-reader access to cuckoo hash tables without trading off space;

- The design of cuckoo filter, a novel data structure based on partial-key cuckoo hashing, which replaces Bloom filters [19] for probabilistic set-membership tests and achieves higher space efficiency and performance while still supporting deletion;

- The integration of our proposed data structures into two practical key-value stores for different workloads: one called MemC3 [41] targets serving transient data from main memory with extremely high throughput and low latency; and the other called SILT [59] is designed to impose extremely low memory overhead to efficiently serve billions of key-value entries on flash from a single node.

Here are the tips on the terminology for readers of this thesis. Caches are studied in different contexts in this thesis. Unless mentioned specifically, we use the term "cache" to refer to an *application-level* cache, and "CPU cache" to the built-in cache inside CPUs. The term "hash function" is also frequently mentioned in different parts of this thesis, but in each part different properties are required. In chapter 2, cryptographic hash functions like SHA-1 [74] are assumed and used in evaluation to ensure collision resistance; while in chapter 3 and chapter 4, we generally apply non-cryptographic hash functions such as BobHash [20] (chapter 3) and CityHash [26] (chapter 4), because they are effective to generate pseudo-random values which are sufficient to build hash tables.

Chapter 2

Randomized Load Balancing in Key-Value Systems

This chapter studies a simple scheme based on caching and randomized load balancing. We show that, for a class of cloud services that mainly read small data without (multi-cell) transactions through front-ends (e.g., key-value queries), caching a very small amount of query results at the frontend can effectively help meet the cluster-wise service-level objectives (SLOs) in terms of handling a target rate of the input queries, regardless of the distribution of these queries.

As data intensive computing grows in both popularity and in scale [24, 33, 36], it becomes simultaneously more important and more challenging to load balance thousands of nodes and beyond. System designers must be ever more careful to ensure their performance does not become bottlenecked due to unevenly partitioned load among cluster nodes. Consistent hashing schemes (e.g., that used in Chord [54, 87]) are therefore popular due to their simplicity and ability to spread data uniformly among nodes. They help balance the static space utilization, but do not balance the dynamic load due to unpredictable shifts in the query workload such as "flash crowds" [11] or adversarial access patterns, either accidentally or as a denial-of-service attack.

This load imbalance problem is particularly critical for architectures with comparatively resource-constrained backends, such as the FAWN-KV system ("Fast Arrays of Wimpy Nodes") [9]. FAWN-KV achieves high energy- and cost-efficiency by using wimpy nodes with slower CPUs, less memory, and solid state drives for storage; on the other hand, these wimpy nodes are particularly susceptible to load imbalance, even at small sizes, because they have less headroom for handling query bursts or popularity shifts.

This chapter investigates the FAWN-KV system and shows that systems like FAWN-KV can effectively prevent their backend nodes from being saturated by using a *popularity-based small front-end cache*, which directly serves a very small number of popular items in front of primary servers ("back-end nodes"). This cache can be surprisingly small and fit in the L3 cache of a fast CPU, enabling an efficient, high-speed implementation compatible with front-end load balancers and packet processors. Intuitively, this small cache works because of the opposition between caching and load balancing: A skew in popularity harms load balance but simultaneously increases the effectiveness of caching. The cache therefore serves the most popular items without querying the back-end nodes, ensuring that the load across the back-end nodes is more uniform. Our result not only applies to FAWN-KV, but also generalizes to an important class of cloud services (e.g., systems with an architecture as shown in Figure 2.1).

The contribution of this work includes

- Derivation of an $O(n \log n)$ lower-bound on the necessary cache size where n is the total number of back-end nodes. The cache size does not depend on the number of items stored in the system.
- Simulation and empirical results to validate our analysis running a key-value storage system on an 85-node cluster.

This chapter focuses on demonstrating how caching helps avoid hotspots in a key-value cluster and ensures high system throughput. The rest of this thesis investigates how to use novel data structures (Chapter 3 and Chapter 4) to implement high-performance (i.e., memory-speed) key-value caches in the frontend and memory-efficient key-value stores in the backend (Chapter 5).

2.1 Target Services

This chapter targets a class of services that are popular building blocks for several distributed systems, with three properties:

1. **Randomized partitioning**. The service is partitioned across cluster nodes and the way the service is partitioned is opaque to clients. (e.g., a key is hashed to select the back-end that serves it.)

2. **Cheap to cache results**. The cache can easily store the result of a query or request and serve future requests without costly recomputation or retrieval.

3. **Costly to shift service**. Moving service from one back-end node to another is expensive in network bandwidth, I/O, and/or consistency and indexing updates. In other words, the partitioning cannot be efficiently changed on the timescale of a few requests.

Systems fitting this category include:

- *Distributed key-value caches* such as memcached [65];

- *Distributed key-value storage systems* such as Dynamo [36], Haystack [14], and FAWN-KV [9];

- *Distributed shared-nothing and non-relational data stores* such as Redis [85] and Aerospike (Citrusleaf) [6].

Services we *do not* consider are those in which:

- *Queries can be handled by any node*, such as a web server farm with a large set of identical nodes, each of which is capable of handling any request. These services do not require caching for effective load-balancing.

- *Partitioning is predictable or correlated*: For example, column stores such as BigTable [24] and HBase [3] store lexicographically close keys on the same server. Our results apply only when keys are partitioned independently—in other words, for systems where a client cannot easily find a large set of keys that would all be sent to the same back-end node.

2.2 System Model and Example

Table 2.1 summarizes the notation used in the system model for analysis.

Model Consider a key-value system such as that shown in Figure 2.1 that serves a total of m distinct items partitioned across n back-end nodes $1, 2, \cdots, n$ where node i can handle at most r_i queries per second. The system caches the c most popular items ($c \leq m$) at a front-end. On a cache hit, the front-end can serve the client request without querying the corresponding back-end server.

Symbol	Meaning
n	total # of back-end nodes
m	total # of key-value pairs stored in the system
c	total # of key-value pairs cached in the frontend
R	sustainable query rate of the entire system
L_i	query rate going to node i
r_i	max query rate supported by node i
p_j	fraction of queries for the jth key-value pair

Table 2.1: Notation used for the analysis.

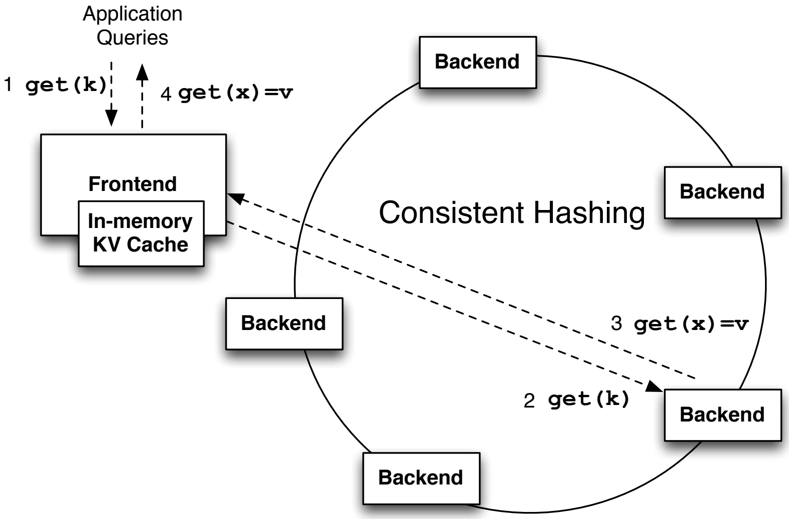


Figure 2.1: Small, fast cache at the front-end load balancer.

Assumptions This analysis makes assumptions about the system. As real systems may not necessarily obey these assumptions, we examine the effect of the factors that may affect load balancing in Section 2.4.

1. **Randomized key mapping to nodes**: each of m keys is assigned to one of the n storage nodes, and this mapping is unknown to clients.
2. **Cache is fast enough**: the front-end is fast enough to handle queries and never becomes the bottleneck of the system throughput.
3. **Perfect Caching**: queries for the c most popular items always hit the cache, while other items always miss the cache.
4. **Uniform Cost**: the cost to process a query at a back-end node is the same, regardless of the queried key or the back-end node processing the query.
5. **Workloads are read-most, or read-write mixed but the cache is write-back**: otherwise, repeatedly updating a single key with a write-through cache can easily saturate its corresponding backend.

Goal: Guaranteed Throughput Our goal is to evaluate the throughput R the system can sustain *regardless of the query distribution.* Load balancing is critical to sustainable throughput, because once any node i becomes saturated (i.e., serving at its full speed r_i), the system *cannot guarantee* more throughput to clients, even though other nodes still have spare capacity. In other words, we are interested in the system throughput even with adversarial query patterns.

Challenge: Adversarial Workloads For clarity of the presentation, we assume the cluster is serving an *adversarial workload*, whose goal is to maximize the chance of saturating one node. The adversarial workload generator knows:

- which m keys are stored on the system;
- the number of back-end servers n; and
- the cache size c.

However, the adversary *cannot easily target one specific node* as the hotspot, because it does not know which keys are assigned to which nodes. We note that we are not considering an intelligent, malicious adversary, who might resort to adaptive attacks to guess where keys sit. We use the blind adversarial model only to generate a worst-case workload to lower-bound the cache size, not to make strong claims about the system's security.

In order to generate a skewed workload on the back-end, the adversarial strategy is to query for x different keys according to some distribution. This workload may cover all m keys, or a specific subset of all keys; it may also request different keys at different probabilities. Formally, an adversarial strategy can be described as a distribution S

$$S = (p_1, p_2, \cdots, p_m), \tag{2.1}$$

where p_i denotes the fraction of queries for the ith key. $p_1+p_2+\cdots+p_m = 1$. Without loss of generality, we list the keys in monotonically decreasing order of popularity, i.e.,

$$p_1 \geq p_2 \geq \cdots \geq p_m.$$

2.2.1 An Example: FAWN-KV

Before presenting our analytical results (Section 2.3), we first introduce our system model through an example—the FAWN-KV [9] key-value storage system—which we use for our experimental results in Section 2.4. FAWN-KV is a distributed high-performance key-value storage system. Like other key-value hash tables such as Dynamo [36], memcached [65], Citrusleaf, and cluster distributed hash tables [47], FAWN-KV provides a simple hash table-like interface for key-value operations:

- `PUT(k,v)` maps the key `k` to the value `v`; and
- `v=GET(k)` retrieves the value `v` associated with key `k`.

Like the diagram in Figure 2.1, a FAWN-KV cluster has one *front-end* node that directs queries from client applications to the appropriate *back-end* storage node by hashing the key being queried for. All keys are stored on the back-end nodes.

Clients for key-value storage services such as FAWN-KV are typically other applications running in the datacenter. These clients often generate a large number of key-value lookups to perform a single user-facing operation such as displaying a web page: for example, Facebook is thought to issue on average 130 internal queries to compose a single page, and Amazon between 100 and 200 [79]. Many of these requests are dependent upon earlier queries, making latency and strict adherence to service-level agreements critical for the performance of the overall enterprise [36].

A Resource-Constrained Testbed The FAWN-KV system was originally designed for "FAWN" clusters, or "Fast Arrays of Wimpy Nodes," which are particularly susceptible to load imbalance, even at small sizes, because the back-end nodes are comparatively resource-constrained. They have slower CPUs, less memory, and use a single solid state drive for storage. This architecture is energy and cost-efficient [9], but has less headroom for handling query bursts or popularity shifts [53]. The lower capacity of the back-ends also allows us to experiment with load balancing strategies using a userland-based cache implementation instead of, e.g., the hardware or specialized network processor implementations used for high-speed commercial load balancers.

Consistent Hashing: Key Ranges to Nodes FAWN-KV organizes the back-end nodes into a storage ring-structure using consistent hashing in a 160-bit ring space (the hashing scheme used in Chord [87]). This consistent hashing is used to partition the key space among different nodes while smoothly handling node arrivals and departures. FAWN-KV does not use Chord's multi-hop routing; instead, the front-end node maintains the entire node membership list and forwards the queries directly to the back-end node containing a particular data item.

Small-Fast-Cache Design and Implementation As we show below, ensuring load balancing requires a relatively small cache; to achieve high throughput, however, the font-end cache must be fast enough to keep the cluster of nodes behind it busy. A contemporary server CPU with several MBs of last-level cache (e.g., L3 cache) can satisfy these two requirements to act as a front-end for a cluster of Atom-based FAWN nodes with SSDs.

In this chapter, we demonstrated the caching effect by a simple software-based front-end cache based on a widely available concurrent hash table implementation [4] that supports thread-safe access from multiple threads. In chapter 5 we will explore an alternative design that is built upon advanced concurrent hashing schemes to support concurrent accesses with higher performance. Our testbed, the FAWN key-value system, uses Thrift [5] for marshaling, unmarshaling, and sending RPCs between the front-end and back-ends.

2.3 Analysis: Why the Hotspots are Avoided

This section presents analytical results showing that a small front-end cache can provide load balancing for n back-end nodes in our target class of systems by caching only $O(n \log n)$ entries, even under worst-case request patterns. The key intuition behind our results is that the cache must merely be large enough to ensure that uncached queries will be spread evenly over the back-end nodes. The surprising effectiveness of a small cache is due to the fact the worst case for load balance—a highly imbalanced query workload—is simultaneously the best case for caching, and vice-versa.

The major results derived in this section include:

- The worst case workload for the system (i.e., the best case for the adversary to create load imbalance) is to send queries for $x > c$ different items at an equal rate for each item, where c is the number of cache entries. (Section 2.3.1)
- Under the adversarial workloads, if the system has a small, fast cache at the front-end of size $c = k \cdot n \log n + 1 = O(n \log n)$, with high probability it maintains reasonably good load balance and avoid hotspots. (Section 2.3.2)

2.3.1 Adversarial Workloads

We first examine the best strategy the adversary could adopt when the back-end servers have homogeneous capacity: $r_1 = \cdots = r_n = r$.

First, when the system has no front-end cache ($c = 0$), the best adversarial strategy is trivial: always query for one arbitrary key, e.g., the first key, to saturate the corresponding storage node. Its distribution is thus:

$$S = (1, 0, \cdots, 0). \tag{2.2}$$

Under this workload, only one node is saturated and the others are completely idle; in other words, the system can satisfy only r queries per second even though its aggregate capacity is $n \cdot r$.

This trivial case demonstrates that without front-end caching the throughput of the system does not scale under an adversarial access pattern.

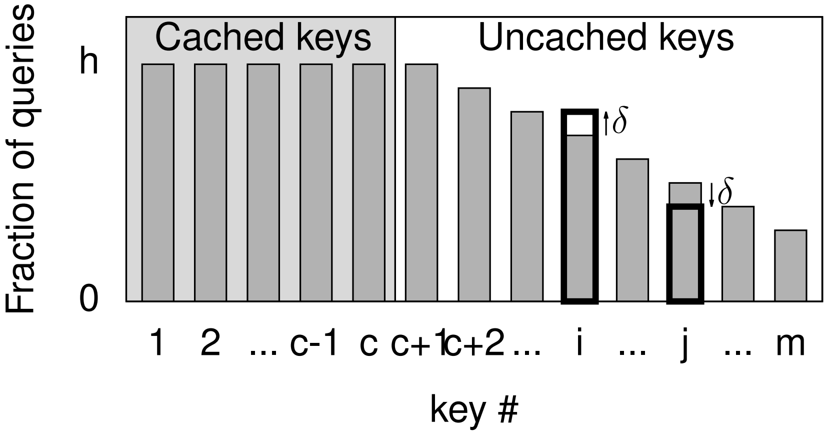


Figure 2.2: The construction of the best strategy for the adversary. The adversary can increase the expectation of the maximum load by moving query rate δ from a uncached key j to a more popular uncached key i (Theorem 1).

When the system has a cache of size $c > 0$, the c most frequently requested keys will all hit the front-end cache:

$$S : \underbrace{p_1 \geq p_2 \geq \cdots \geq p_c}_{\text{cached keys}} \geq \underbrace{p_{c+1} \geq \cdots \geq p_m}_{\text{uncached keys}} . \tag{2.3}$$

To create back-end hotspots, the adversary must therefore query more than c keys in order to bypass the cache and hit the back-ends. In these c cached keys, the adversary does not benefit from querying one key (e.g., key i) at a higher rate than any other cached key; the adversary will benefit from making fewer queries for key i and more for some uncached key(s). Therefore, the adversary should always query the first c keys (which will be cached) at the same probability (i.e. $p_1 = p_2 = \cdots = p_c$):

$$S : \underbrace{p_1 = p_2 = \cdots = p_c}_{\text{cached keys}} = h \geq \underbrace{p_{c+1} \geq \cdots \geq p_m}_{\text{uncached keys}} \geq 0. \tag{2.4}$$

The following theorem states the best strategy for the adversary in terms of the uncached keys (see Figure 2.2):

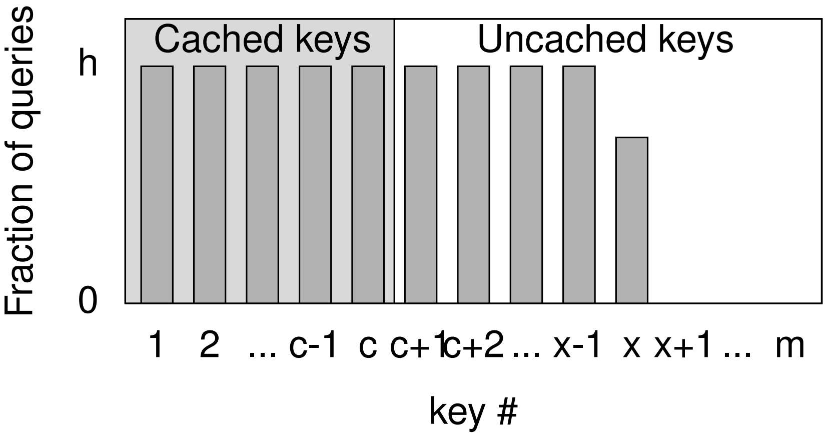


Figure 2.3: The constructed best strategy for the adversary to maximize load imbalance. The adversary queries x keys, where the rate for $x-1$ is the same.

Theorem 1 *If any distribution S has two uncached keys i and j such that $h > p_i \geq p_j > 0$, the adversary can always construct a new distribution S' based on S to increase the expectation of L_{max}. This new distribution S' is the same as S except $p'_i = p_i + \delta$, $p'_j = p_j - \delta$ where $\delta = \min\{h - p_i, p_j\}$.*

Proof Assume the adversary is sending R queries per second. The only difference between strategies S and S' is that S' increases the query rate for key i by $\Delta = \delta \cdot R$ and decreases the query rate for key j by Δ.

If key i and j are mapped to the same storage node, it is trivial that S and S' generate the same load at the node.

If key i is served by node u and key j by node v, there are four cases:

- **case 1: node u has the highest load, node v has less.** Because node u is the most loaded node under S, shifting Δ load from node v to node u keeps node u the most loaded—and increases its load by Δ. The highest load under S', denoted by $L'_{\max}$, is

$$L'_{\max} = L_u + \Delta = L_{\max} + \Delta.$$

- **case 2: node v has the highest load, node u has less.** By shifting Δ load from node v to node u, the max load under S' will be decreased. However, the decrease of $L'_{\max}$ is no more than Δ because the load of node v by S' is at least $L_{\max} - \Delta$.

$$L'_{\max} \geq L'_v = L_v - \Delta = L_{\max} - \Delta$$

- **case 3: neither node u nor v has the highest load.** In this case, reducing the load of node v does not decrease the load of the original most loaded node. Increasing the load of node u by Δ may, however, make node u the most loaded. As a result, in this case, S' is at least as good as S for the adversary:

$$L'_{\max} = \max\{L_u + \Delta, L_{\max}\} \geq L_{\max}$$

- **case 4: both u and v have the same max load.** The max load by S' will increase by Δ because

$$L'_{\max} = L_u + \Delta = L_{\max} + \Delta.$$

In case 1 and case 4, $L'_{\max}$ is increased by Δ; while in case 2, it is decreased by at most Δ. Note that node u and v are both randomly chosen by the hashing from the pool of n nodes. Therefore, node u, serving key i with a higher query rate (i.e., $p_i > p_j$), has a better chance to become the most loaded node than node v serving key j. In other words,

$$\mathbb{P}\{\text{case1}\} \geq \mathbb{P}\{\text{case2}\}. \tag{2.5}$$

In terms of expectation, the max load $L'_{\max}$ is then increased by S' because

$$\begin{aligned} & \mathbb{E}[L'_{\max}] - \mathbb{E}[L_{\max}] = \mathbb{E}[L'_{\max} - L_{\max}] \\ \geq \; & \Delta \cdot (\mathbb{P}\{\text{case1}\} - \mathbb{P}\{\text{case2}\} + \mathbb{P}\{\text{case4}\}) \geq 0 \quad \blacksquare \end{aligned}$$

Intuitively, the theorem suggests that the adversary should always shift some load from key j to a more queried key i until this key gets the same fraction as the cached keys. If the adversary applies this process repeatedly, it ends up with a distribution with an equal probability for the first $x - 1$ keys (where x is the number of keys it queries for), or

$$\begin{aligned} S: \quad & \underbrace{p_1 = \cdots = p_c}_{\text{cached keys}} = h = \underbrace{p_{c+1} = \cdots = p_{x-1} \geq p_x}_{\text{uncached keys}} > 0 \\ & p_{x+1} = \cdots = p_m = 0 \end{aligned} \tag{2.6}$$

In other words, the best strategy for the adversary is to query the first $x-1$ keys at probability h and the last one at probability $1-(x-1)h$ ($\sum p_i = 1$), as illustrated in Figure 2.3.

Note that Eq. (2.6) does not state the value of x or h. Intuitively, the adversary has two concerns when choosing x. First, since a fraction c/x of the total load will be served by the cache, x should be large enough to ensure enough load bypasses the cache. Second, x cannot be too large otherwise the query load covers a large number of keys uniformly—the best case the system can expect.

2.3.2 Throughput Lower Bound with Cache

Since each key is assigned to one of the n nodes randomly, the load on each of the nodes can be bounded by the well-known *Balls-in-Bins* model [69, 84]. Imagine we are throwing M identical balls into N bins and each bin is picked randomly. When $M \gg N \log N$, the number of balls in any bin is bounded by

$$\frac{M}{N} + \alpha\sqrt{2 \cdot \frac{M}{N} \cdot \log N}, \tag{2.7}$$

with high probability (i.e., $1 - O\left(\frac{1}{N}\right)$). $\alpha > 1$ is a constant factor affecting the confidence.

In our model, there are $(x-c)$ uncached keys that can be considered as the balls and n servers as the bins. Setting $N = n$, $M = x - c$ in Eq (2.7), the number of different keys served by any server is bounded by

$$\frac{x-c}{n} + \alpha\sqrt{\frac{2(x-c)}{n}\log n}. \tag{2.8}$$

Based on Eq. (2.8), we can derive the bound for the load imposed on a back-end server. If the adversary is sending queries at rate R, for each key the query rate is at most $R/(x-1)$. Given the maximum number of keys served by any node (Eq. (2.8)), an upper bound of the expected load on each node is

$$\begin{aligned}\mathbb{E}[L_{\max}] &\leq \left(\frac{x-c}{n} + \alpha\sqrt{\frac{2(x-c)}{n}\log n}\right) \cdot \frac{R}{x-1} \\ &= \left(\frac{x-c}{x-1} + \alpha\sqrt{\frac{2(x-c)}{(x-1)^2} n \log n}\right)\frac{R}{n}\end{aligned} \tag{2.9}$$

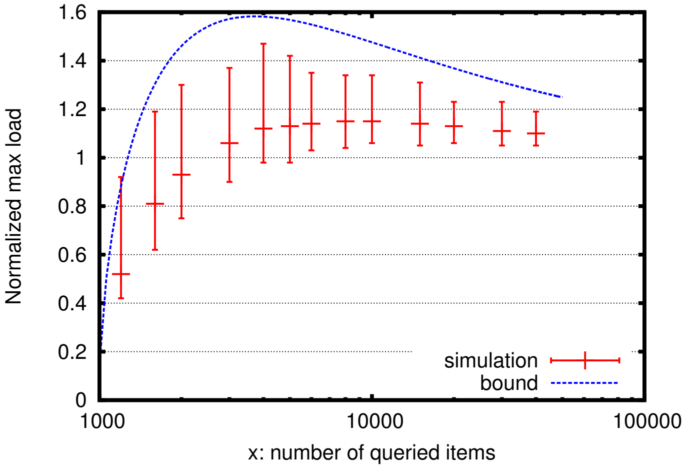
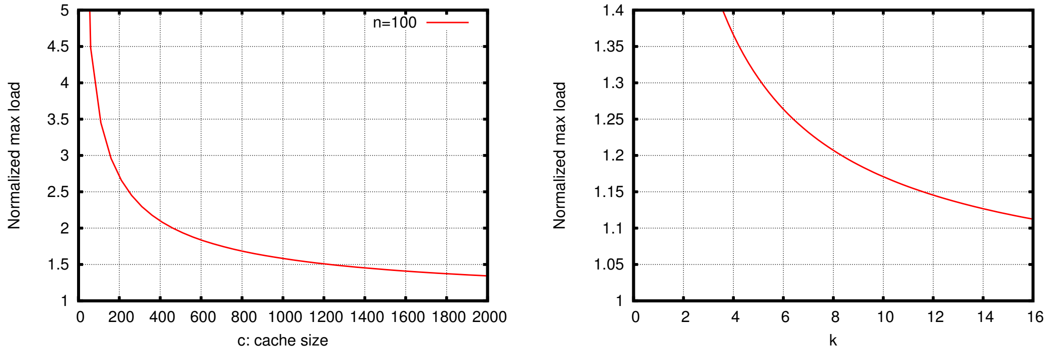


Figure 2.4: Simulation of the maximum number of keys on $n = 100$ nodes with cache size $c = 1000$, with 1000 runs.

To examine the accuracy and the tightness of this bound, we simulate a system with 100 nodes and a cache of size 1000. For each run of the simulation, x ($x > 1000$) different keys are queried at the same rate, and the load of the most loaded back-end node is recorded. As shown in Figure 2.4, each bar in represents the maximum load observed from 1000 independent runs with average, max, and min. The curve is calculated from Eq. (2.9) with $\alpha = 2$. The figure shows the bound has a small gap from the numerical results when $\alpha = 2$. Figure 2.4 also shows that there is a global maximum point achieved by some value x. This maximum point is the best any adversary can achieve given the cache size c and number of nodes n.

Eq. (2.9) is the maximum possible (at high probability) load the adversary can impose on any back-end nodes. Based on c and n, the adversary can set x to a proper value—a value not so large as to make the query load too even, and not so small as to hit cache too often—to maximize this possible load. By optimizing Eq. (2.9), we have the maximizer of Eq. (2.9) to be

$$x^* = 1 + 2(c-1)\left(1 + \frac{1}{\sqrt{1 + \frac{2\alpha^2 n \log n}{c-1}} - 1}\right). \tag{2.10}$$



(a) vary the cache size c

(b) vary the factor k

Figure 2.5: Normalized maximum load

Based on Eq. (2.10), we can bound Eq. (2.9) for the worst case (the maximum point in Figure 2.4) for any x by

$$\mathbb{E}[L_{\max}] \leq \frac{1+\sqrt{1+2\alpha^2\frac{n\log n}{c-1}}}{2}\cdot\frac{R}{n}, \tag{2.11}$$

which leads to the normalized throughput for the most loaded node being bounded by

$$\frac{\mathbb{E}[L_{\max}]}{R/n} \leq \frac{1+\sqrt{1+2\alpha^2\frac{n\log n}{c-1}}}{2}. \tag{2.12}$$

Figure 2.5(a) illustrates the relationship between the max load (normalized) calculated in Eq. (2.12) and the cache size c. Note that increasing the cache size beyond a certain point provides diminishing returns, which suggests how to set the cache size in order to bound the maximum normalized load seen by any one back-end.

Cache of Size $\boldsymbol{O(n\log n)}$ If we choose a cache size of $c = k\cdot n\log n + 1$ where k is a constant factor, the load bound shown in Eq. (2.12) becomes constant in the system size:

$$\frac{1}{2}\left(1+\sqrt{1+\frac{2\alpha^2}{k}}\right) \tag{2.13}$$

Figure 2.5(b) shows the normalized load (Eq. (2.13) as a decreasing function of the constant factor k for cache size. Note that the normalized load is highly sensitive to k when k is small. When $k = 8$, its value is about 1.2 which means the most loaded node gets at most 20% more work to do than the average amount of work. When k further increases, the decrease of the load is diminishing.

Because Eq. (2.13) is independent of n, a system designer can choose a value for k that bounds the load (amount of over-provisioning required) at the back-ends. This normalized load/over-provisioning will be the same for any n, provided the front-end cache is scaled appropriately (to the value of c given above).

Throughput Lower Bound If the capacity r of each node is larger than the upper bound of $\mathbb{E}[L_{\max}]$ given in Eq. (2.11), then with high probability, the adversary can never saturate any node. The system can therefore guarantee that its throughput will always be equal or better than R queries per second.

$$R \geq \frac{2}{1 + \sqrt{1 + 2\alpha^2 \frac{n \log n}{c-1}}} \cdot n \cdot r, \tag{2.14}$$

no matter what distribution of key queries the adversary uses.

2.3.3 Heterogeneous Servers

So far we have assumed homogeneous nodes in the cluster so that each back-end can serve requests at the same maximum rate. In practice, however, nodes in the cluster tend to be heterogeneous. For example, nodes could belong to different generations and the latest nodes usually perform better than the older nodes.

A common way to address heterogeneity is to partition the service among nodes according to their capacity [30, 54, 87]. Each physical node hosts multiple virtual nodes, and each virtual node acts as an independent node in the cluster. By assigning more virtual nodes to the servers of higher capacity, the system can balance the capacity and the workload among heterogeneous nodes.

To measure the load for heterogeneous nodes, we normalize the load for each node by its capacity, where the lowest-capacity nodes have only one virtual node. Assume there are still n physical nodes as before and each hosts v virtual nodes on average. Therefore, there are $n' = vn$ virtual nodes in total. Applying Eq. (2.11), the highest load among all virtual nodes is bounded by:

$$\frac{1+\sqrt{1+2\alpha^2\frac{n'\log n'}{c-1}}}{2}\cdot\frac{R}{n'}, \tag{2.15}$$

Assume the most loaded physical node hosts z virtual nodes; its normalized load is bounded by:

$$\frac{\left(\frac{1+\sqrt{1+2\alpha^2\frac{n'\log n'}{c-1}}}{2}\cdot\frac{R}{n'}\right)z}{\frac{z}{n'}R}=\frac{1+\sqrt{1+2\alpha^2\frac{n'\log n'}{c-1}}}{2}, \tag{2.16}$$

which is just the maximum normalized load for the virtual nodes.

As a result, as long as we scale the factor k by a factor $(v+v\log_n v)$, or :

$$c=(v+v\log_n v)k\cdot n\log n+1, \tag{2.17}$$

the maximum load of any physical node is still bounded by

$$\frac{1}{2}\left(1+\sqrt{1+\frac{2\alpha^2}{k}}\right). \tag{2.18}$$

The end result is that the front-end cache can provide effective load balance for a cluster of heterogeneous capacity if we increase its size by a factor of $(v+v\log_n v)$. Intuitively, this means that the increase in the cache size is proportional to the disparity between the average node capacity and that of the weakest nodes in the system.

2.4 Evaluation on FAWN-KV

We perform experiments on our FAWN-KV cluster to evaluate the effectiveness of load balancing with a front-end cache. Our goals are, first, to validate that load balancing in the real system matches the theory; second, to validate that the performance of the system improves in tandem with the improvement in load balance; and third, to validate that this caching design can operate at high speed.

	Front-end node	Back-end node
CPU:	2× Intel Xeon L5640	Intel Atom D510
Clock:	2.27 GHz	1.66 GHz
# cores:	2×6	2
CPU cache:	2×12 MiB (L3)	512 KiB (L2)
DRAM:	2×24 GiB	1 GiB

Table 2.2: Specifications of front- and back-end nodes in FAWN-KV testbed

2.4.1 Experiment Overview

This section describes the cluster hardware and common experimental parameters.

Experimental Testbed Our FAWN-KV cluster consists of one high-performance front-end node and 85 low-power back-end nodes. The front- and back-end node's specifications are shown in Table 2.2. All nodes are connected to a switch; the front-end node uses a 10 GbE link, while back-end nodes use 1 GbE links. The network is never the bottleneck in our experiments.

Workload Generation Our experiments use synthetic key-value pair operations. In all experiments, a client first generates and puts $m = 8.5$ million key-value pairs into the cluster; thus, on average, each of the $n = 85$ back-end nodes is responsible for serving approximately $100,000$ unique key-value pairs. The mapping of a given key-value pair to a back-end is done by hashing the key.

For query generation, the client selects x different keys ($x \leq m$) from all generated keys with a certain access pattern and popularity distribution as we describe for each experiment. The client pipelines queries to hide network latency, but to keep latency within a reasonable value, limits the maximum number outstanding queries to 1000 keys per back-end.[1] The client resides on the same physical machine as the front-end node. The key size is 20 bytes, and the value size is 128 bytes. In this section, all workloads generated and evaluated are read-only.

Unless otherwise specified, the following experiments use all 85 back-end nodes. Because not every back-end node is equipped with an SSD, we first measured the throughput of a single node serving queries from its SSD, which it can do at approx-

[1] The nodes serve roughly 10,000 queries/second, so a queue of 1,000 queries adds at most 100ms of latency.

imately $10,000$ queries/second. We then emulate the SSD I/O behavior by having the back-ends serve data from a rate-limited memory-based disk that serves 10,000 requests/second, to be able to scale our experiments to more nodes than we have SSDs for.

2.4.2 Caching Effect: Cluster Scaling Under Different Workloads

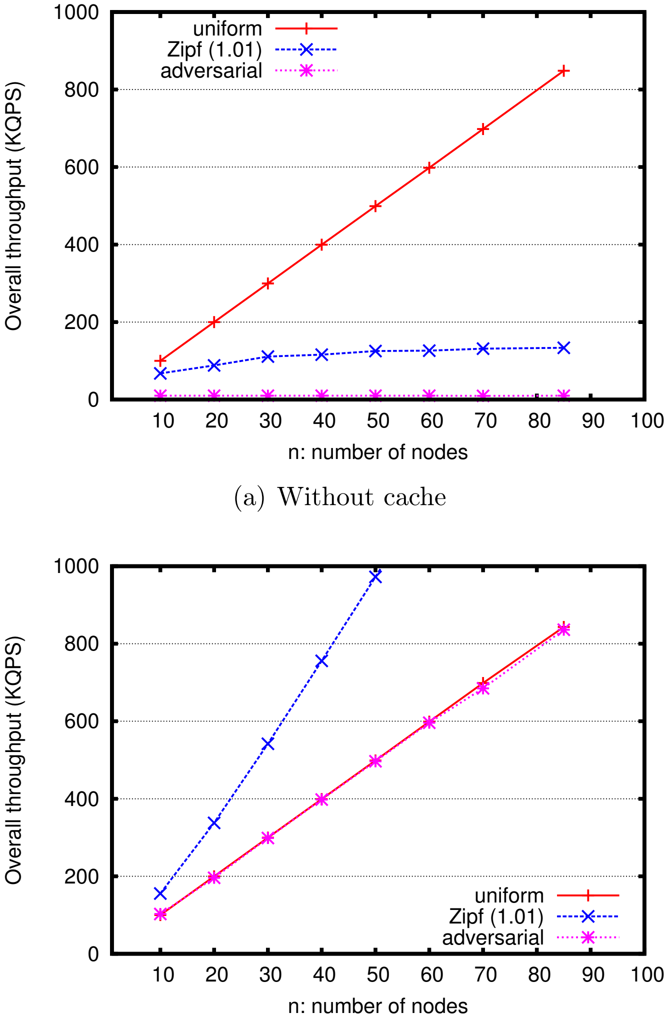
Figure 2.6 compares the read throughput achieved by our FAWN-KV testbed cluster, *without* and *with* a front-end cache deployed. The size of the cluster increases from 10 nodes to 85 nodes. We explore the scalability of system throughput with three different access patterns:

- a uniform distribution across all $m = 8.5$ million keys, which is expected to be a good case of load balancing, serving as a baseline;
- a Zipf distribution with parameter 1.01, which has a bias towards a few keys but also has a heavy tail[2]; and
- an adversarial access pattern, which is obtained by varying the number of selected keys (x) to find the worst-performing value of x, and querying at random for only those keys.

Figure 2.6(a) shows the case when no cache is used at the front-end. The throughput of the uniform workload scales linearly as the number of nodes grows, with each back-end node serving 10 K queries/second. The throughput of the Zipf workload, however, grows slowly and has diminishing returns with each additional node. With Zipf, the workload is biased to a small set of keys, and the nodes serving these keys become a bottleneck, limiting the overall throughput of the cluster. The adversarial access pattern achieves the worst throughput. In the no caching case, this pattern queries only one key regardless of the size of the cluster, and thus always has a total throughput of 10 K queries/second.

Figure 2.6(b) shows the throughput when using a small front-end cache of size $c = 8 \cdot n \log n + 1$, where n is the number of back-end nodes. The throughput for the random workload remains almost the same as before because the cache is relatively small compared to the working set size; the cache absorbs a small number of requests,

[2] Zipf and other heavy tail distributions often better characterize real-world workloads.



(a) Without cache

(b) With cache

Figure 2.6: Read throughput of our FAWN-KV cluster with n back-end nodes where n increases from 10 to 85, under different access patterns including uniform, Zipf, and an adversarial workload.

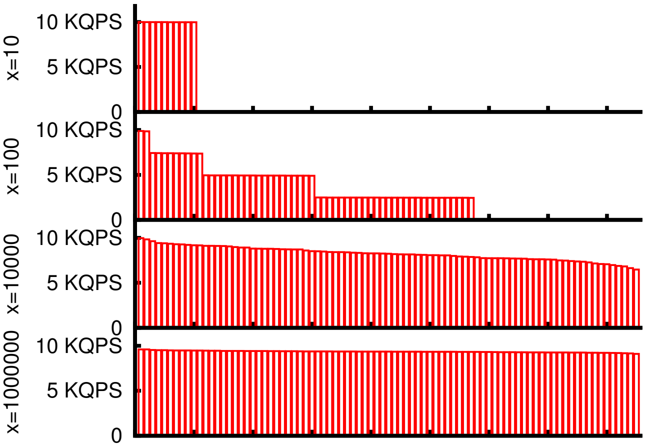


Figure 2.7: Throughput of each back-end node for different values of x when $c = 0$. Node IDs (x-axis) are sorted according to their throughput.

but most of the queries are distributed evenly across the back-end nodes. Zipf's bias towards a small number of keys benefits most from having a font-end cache—even a very small one—the system performance even exceeds the aggregate raw throughput that the back-end nodes can provide. Finally, the system performance for the adversarial access pattern matches the theoretical results: an appropriately-sized front-end cache (based on the number of nodes in the given trial) produces the same performance as did the uniform distribution. With the front-end cache, *all* workloads achieve at least the linear scaling of the purely uniform workload.

2.4.3 Load (Im)balance Across Back-End Nodes Without Cache

To further understand the interaction between load balancing and system throughput, Figure 2.7 shows a snapshot of the individual node performance with front-end caching disabled under the uniform random workload. We show performance when querying for four different set sizes (values of x).

- When $x = 10$, the number of active keys is smaller than the cluster size, and thus only 10 back-end nodes serve queries. The aggregate performance is therefore quite poor. The nodes operate at the same rate as each serves only one key.

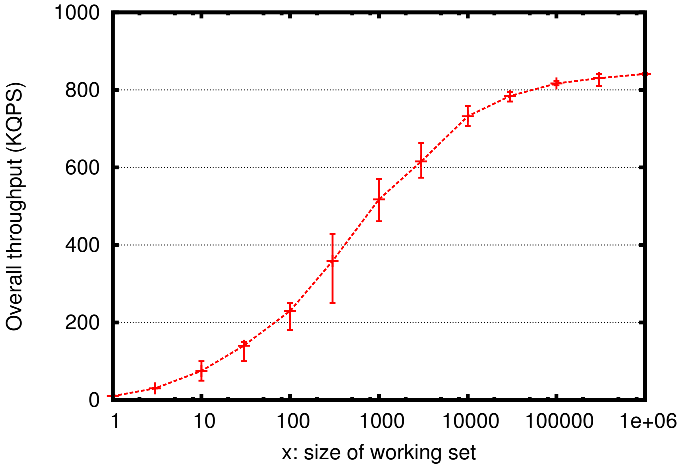


Figure 2.8: Throughput that the adversary can obtain by querying for different numbers of keys when the cluster has 85 back-end nodes and no cache is used.

- When $x = 100$, 28 of the 85 nodes remain idle: with the random key-to-node assignment, some nodes handle four or more keys while others handle zero. The overall load balancing—and performance—is still poor.

- When $x = 10000$, the working set is much larger than the cluster size, and all back-ends are used; however, the load distribution remains skewed, which reduces the overall throughput.

- When $x = 100000$, the load is distributed almost perfectly.

In summary, we see empirically that, without a cache, load balancing is very susceptible to the working set size x, and there are three operating regions for x. If x is smaller than the number of back-ends, not all of the back-ends are used, and the performance suffers. But, even if x is larger than the number of back-ends, the load distribution can be uneven (due to the balls-in-bins game), and the performance is sub-optimal. Only when the number of unique objects queried is sufficiently greater than the number of back-ends does the system achieve good load balance across all of the nodes.

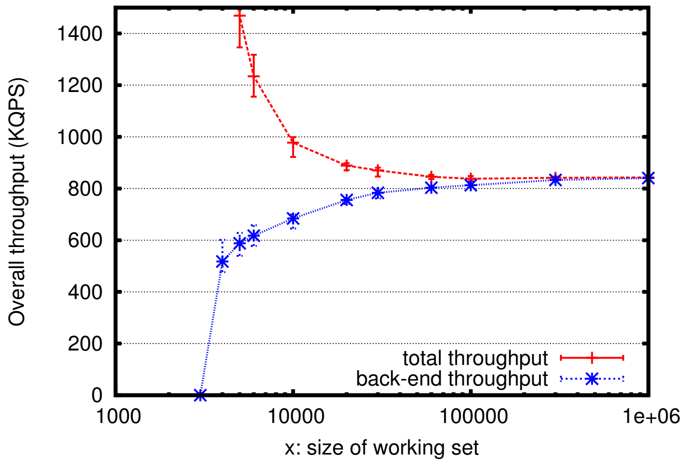


Figure 2.9: Throughput that the adversary can obtain by querying for different numbers of keys when the cluster has 85 back-end nodes and $c = 3000$. Results for $x < 3000$ are omitted because all queries would be served by the front-end cache.

2.4.4 Adversarial Throughput Under Different Query Patterns

Figure 2.8 shows the overall system throughput *without* a front-end cache. On the x-axis, we vary the number of unique keys that the client requests, x, from 1 to 1 million. Without caching, $x = 1$ (i.e., always querying one key) gives the worst system throughput since all queries go to a single back-end node. The system throughput increases with x as the query load is distributed more evenly across the back-ends, and this performance gain with larger x diminishes as the back-end nodes operate at nearly full capacity.

In contrast, Figure 2.9 shows the breakdown of the throughput *with* a front-end cache. The cache is sized based upon our theoretical results ($8 \cdot n \log n + 1$). The bottom curve shows the queries/second served exclusively by the back-end nodes—the trend is similar to that in the no-caching case. The top curve shows the *total* throughput being served by the cache plus the back-end nodes. The contribution of the front-end cache diminishes as x grows, with the aggregate throughput converging eventually to the back-end capacity.

In summary, the back-end throughput alone is low when x is small, as the load is not perfectly balanced as shown in Figure 2.8. In this region, however, a small cache is very efficient to prevent hotspots and ensure enough good performance. When x is

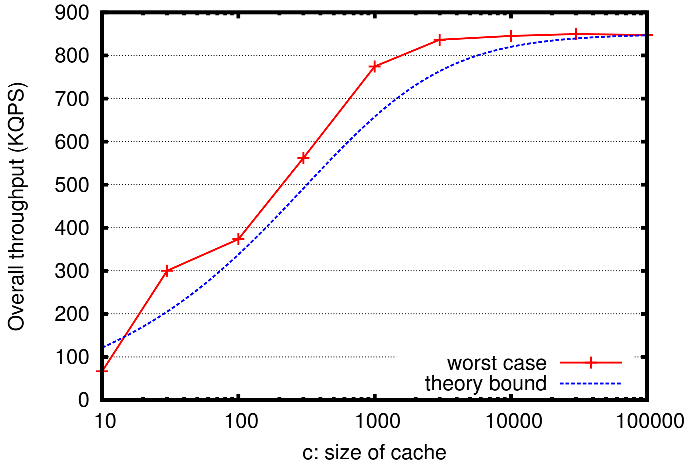


Figure 2.10: Adversarial throughput as the cache size increases. The "worst case" line shows the worst throughput achieved when testing ten different sizes of adversarial request sets.

larger, the caching effect shrinks, but the back-end throughput grows rapidly. These two effects combine to guarantee high performance regardless of the number of keys queried for.

Cache Size vs. Worst Case Performance Figure 2.10 shows the relationship between cache size and worst-case system performance (the best the adversary can do). For each cache size on the x-axis, we test ten different values of x (number of unique keys requested) to find that which produces the worst throughput. The total number of back-ends is fixed at 85 nodes. The figure shows the experimental results and the theoretical lower bound. As expected, the measured throughput is higher than the prediction, except for the smallest cache size (in which case the balls-in-bins approximation we made in our analysis no longer holds).

2.5 Related Work

Distributed Key-Value Storage Systems Dynamo [36] is a distributed key-value storage system for high availability based on consistent hashing [30, 54, 87] and vector clocks [57]. Although Dynamo employs a load balancing strategy similar to virtual nodes [30, 54, 87], its empirical study shows 10% of nodes have at least 15%

higher load than the average load almost all the time, implying that load imbalance is a persistent problem in the storage cluster.

FAWN-KV [9] utilizes a cluster of slower, energy-efficient storage nodes to reduce system operation costs while providing high performance comparable to conventional storage systems. Since a FAWN cluster tends to consist of a relatively larger number of nodes, good load balancing in FAWN-KV is a key to high aggregated performance. Although the original FAWN-KV system used a front-end cache as an optimization to achieve better load balancing, this chapter formally investigates this problem in the context of FAWN-KV, and provides a deeper understanding of its actual benefits analytically and empirically.

Caching in Distributed Systems In addition to the related work discussed in the previous sections, substantial prior work has examined the use of caching to improve throughput and latency in distributed systems. For example, Markatos examined the relationship between cache size and performance gain when caching search engine results [64]. This work, and much that is similar to it, focuses on improving performance by reducing redundant work. In contrast, our work improves performance by preventing load imbalance from allowing individual cluster nodes to become under-utilized while others are straining. As a consequence, our work is able to demonstrate a large performance boost using a small cache. In contrast, systems such as Facebook are thought to cache over 90% of their data [79] in massive farms of memcached [65] servers, and Google maintains its entire index in tens to hundreds of terabytes of DRAM [31].

Caching for Load Balancing Server-based caching [18] achieves better load balancing by replicating a small amount of data from highly loaded servers to proxies that are close to data requesters. It shares the same framework as our caching in that it uses a "front end" cache which serves requests for popular items without contacting cluster nodes. Server-based caching assumes an exponential popularity model for analysis, whereas our work uses adversarial access patterns that incur a worst-case load distribution, which is important for defining and meeting service level agreements.

Chapter 3

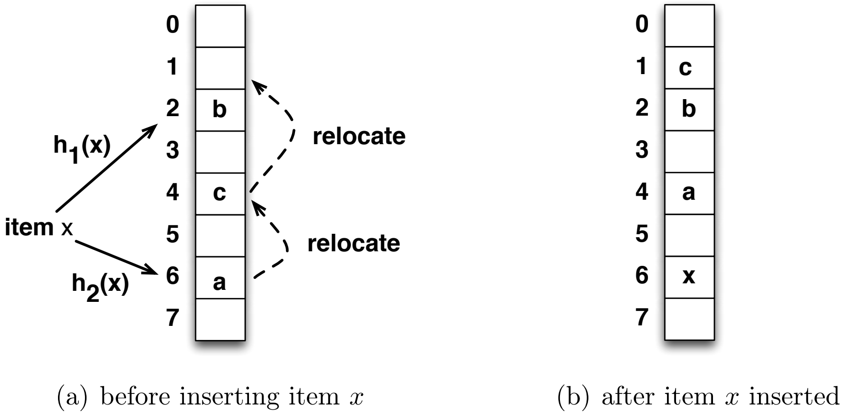
Engineering Cuckoo Hashing

This chapter investigates the algorithm and implementation of *high-speed* and *space-efficient* hash tables that can be used to store the mapping from arbitrary keys to values. Conventional hash tables do not achieve high performance and high space utilization at the same time. For example, hash tables using linear probing waste 50% or more table space; hash tables using chaining create linked-lists for all items assigned to the same table address, and thus incur multiple CPU cache misses when traversing the linked-list on lookup; recently proposed cuckoo hash tables provide space efficiency [1], however, they require multiple memory references for each insertion and lookup, and more importantly, supporting concurrent access to these tables is challenging.

To overcome these limitations, this thesis proposes *optimistic cuckoo hashing*, a novel scheme to build compact, concurrent and CPU cache-aware hash tables. It extends standard cuckoo hashing and achieves three advantages compared to other schemes:

- It achieves high memory efficiency (e.g., 95% table occupancy);
- It supports highly concurrent accesses to the hash table for read-intensive workloads; and
- It keeps hash table operations CPU cache-friendly.

[1] The standard cuckoo hashing as proposed in [81] ensures 50% space utilization, which can be improved to over 90% by using a 4-way (or higher) set associative hash table [38].



(a) before inserting item x (b) after item x inserted

Figure 3.1: Illustration of cuckoo insertion

To our knowledge, optimistic cuckoo hashing is the first scheme that supports highly concurrent access to cuckoo hash tables while still retaining the high space utilization of its sequential version. Our contributions include the parallelized cuckoo hashing algorithm as well as a set of hardware- and workload-inspired optimizations:

- a technique called *partial-key cuckoo hashing* that significantly speeds up cuckoo hash table operations by improving their CPU cache locality;
- an optimistic versioning scheme that supports multiple-reader/single-writer concurrent access to cuckoo hash tables, while preserving the high space benefits; and
- an optimization for cuckoo hashing insertion that improves throughput.

Because of the significant benefits in both performance and space, our proposed scheme perfectly suits the need in building high-performance key-value cache as required in Chapter 2. Chapter 5 therefore investigates how to use it as a basic building block in key-value stores. It also enables another novel data structure called cuckoo filter which will be described in Chapter 4.

3.1 Background: Standard Cuckoo Hashing

Cuckoo hashing [81] is a hashing method that yields highly space-efficient hash tables. A basic cuckoo hash table consists of an array of buckets where each key x has two candidate buckets determined by two hash functions $h_1(x)$ and $h_2(x)$. The lookup procedure is as simple as checking both buckets and returning the value if x is found in either bucket. Insert is more complicated than lookup, so we use an example in Figure 3.1(a) to show how to insert a new item x in to a hash table of eight buckets. If either of x's two buckets (buckets 2 or 6 in this example) is empty, the algorithm inserts x to that free bucket and the insertion completes. If neither bucket has space, as is the case in this example, the item selects one of the candidate buckets (e.g., bucket 6), kicks out the existing item (in this case a) and re-inserts this victim item to its own alternate location. In our example, displacing a triggers another relocation that kicks existing item c from bucket 4 to bucket 1. This procedure may repeat until a vacant bucket is found as illustrated in Figure 3.1(b), or until a maximum number of displacements is reached (e.g., 500 times in our implementation). If no vacant bucket is found, this hash table is considered too full to insert. Although cuckoo hashing may execute a sequence of displacements, its amortized insertion time is still $O(1)$ [81].

Cuckoo hashing ensures high space occupancy because it refines its earlier item-placement decisions when inserting new items. Most practical implementations of cuckoo hashing extend the basic description above by using buckets that hold multiple items, as suggested in [37]. The possible high space utilization by using k hash functions (assuming all hash functions are perfectly random) and buckets of size b is analyzed [44]; and it is also confirmed in experiments that with proper configuration of cuckoo hash table parameters, the table space can successfully be 95% filled [38].

3.2 Partial-Key Cuckoo Hashing

This section describes partial-key cuckoo hashing, a technique that uses a short summary of each key to speed up the *single-thread performance* of cuckoo hash tables. It is also the core technique that enables our proposed cuckoo filter as we will discuss in Chapter 4.

Hash Table Overview Our hash table, as shown in Figure 3.2, consists of an array of *buckets*, each having four *slots*. Our hash table is 4-way set-associative, because without set-associativity, basic cuckoo hashing allows only 50% of the table entries to be occupied before unresolvable collisions occur. Each slot contains a *pointer* to the key-value object and a short summary of the key called a *fingerprint*. This pointer is used to support keys of variable length, because the full keys and values

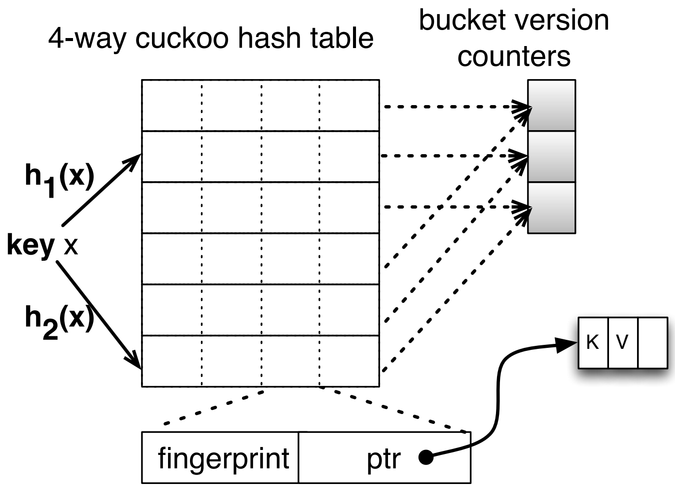


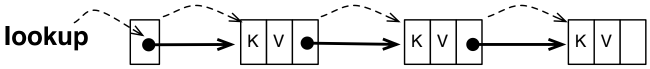
Figure 3.2: Overview of our hash table.

are not stored in the hash table, but stored with the associated metadata outside the table and referenced by the pointer. A null pointer indicates this slot is not used. The fingerprint is used to reduce CPU cache misses when operating on cuckoo hash tables (see explanation later in this section). Figure 3.2 also shows that an array of counters associated with each bucket. They are used to support concurrent access of multiple threads, which will be described in section 3.3.

On `Lookup`, the hash table returns a pointer to the relevant key-value object, or "does not exist" if the key cannot be found. On `Insert`, the hash table returns *true* on success, and *false* to indicate the hash table is too full.[2] `Delete` simply removes the key's entry from the hash table. We focus on `Lookup` and `Insert` as `Delete` is very similar to `Lookup`.

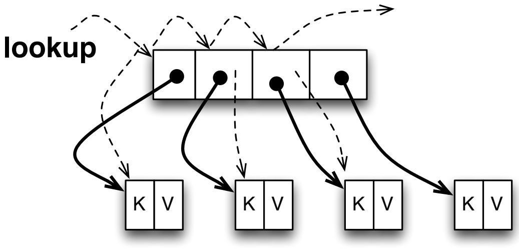
3.2.1 Lookup

Looking up items in a chaining hash table is not CPU cache-friendly. It requires multiple *dependent* pointer dereferences to traverse a linked list:



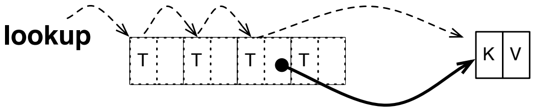
[2] As in other hash table designs, an expansion process can increase the cuckoo hash table size to allow for additional inserts.

Neither is basic cuckoo hashing CPU cache-friendly when keys are stored outside the hash table and referenced by the pointers. This is because checking two buckets on each Lookup makes up to eight (parallel) pointer dereferences:



In addition, displacing each key on Insert also requires a pointer dereference to calculate the alternate location to swap, and each Insert may perform several displacement operations.

Our hash table eliminates the need for pointer dereferences in the common case. We compute a short fingerprint (e.g., one or two bytes) as the summary of each inserted key, and store the fingerprint in the same bucket as its pointer. Lookup first compares the fingerprint, then retrieves the full key only if the fingerprint matches. This procedure is as shown below (T represents the fingerprint)



It is possible to have false retrievals due to two different keys having the same fingerprint, so the fetched full key is further verified to ensure it was indeed the correct one. With a 1-byte fingerprint by hashing, the chance of fingerprint-collision is only $1/2^8 = 0.39\%$. After checking all 8 candidate slots, a negative Lookup makes $8 \times 0.39\% = 0.03$ pointer dereferences on average. Because each bucket fits in a CPU cacheline (usually 64-byte), on average each Lookup makes only 2 parallel cacheline-sized reads for checking the two buckets plus either 0.03 pointer dereferences if the Lookup misses or 1.03 if it hits.

3.2.2 Insert

We also use the fingerprints to avoid retrieving full keys on Insert, which were originally needed to derive the alternate location to displace keys. For an item x, our hashing scheme calculates the indexes of the two candidate buckets as follows:

$$\begin{aligned} h_1(x) &= \text{hash}(x), \\ h_2(x) &= h_1(x) \oplus \text{hash}(x\text{'s fingerprint}). \end{aligned} \tag{3.1}$$

The xor operation in Eq. (3.1) ensures an important property: $h_1(x)$ can also be calculated using the same formula from $h_2(x)$ and the fingerprint. In other words, to displace a key originally in bucket i (no matter if i is $h_1(x)$ or $h_2(x)$), we directly calculate its alternate bucket j from the current bucket index i and the fingerprint stored in this bucket by

$$j = i \oplus \text{hash}(\text{fingerprint}). \tag{3.2}$$

Hence, an insertion only uses information in the table, and never has to retrieve the original item x.

In addition, the fingerprint is hashed before it is xor-ed with the index of its current bucket to help distribute the items uniformly in the table. If the alternate location is calculated by "$i \oplus$ fingerprint" without hashing the fingerprint, the items kicked out from nearby buckets will land close to each other in the table, if the size of the fingerprint is small compared to the table size. For example, using 8-bit fingerprints the items kicked out from bucket i will be placed to buckets that are at most 256 buckets away from bucket i, because the xor operation will alter the eight low order bits of the bucket index while the higher order bits do not change. Hashing the fingerprints ensures that these items can be relocated to buckets in an entirely different part of the hash table, hence reducing hash collisions and improving the table utilization.

Note that, while the values of h_1 and h_2 calculated by Eq. (3.1) are uniformly distributed individually, they are not independent of each other (as required by standard cuckoo hashing) because of the dependence on the fingerprint. This may cause more collisions, and in particular previous analyses for cuckoo hashing (as in [37, 44]) do not hold. Chapter 4 provides a simple analysis on the relationship between achieved load factor and fingerprint size, however, a full analysis of partial-key cuckoo hashing remains open.

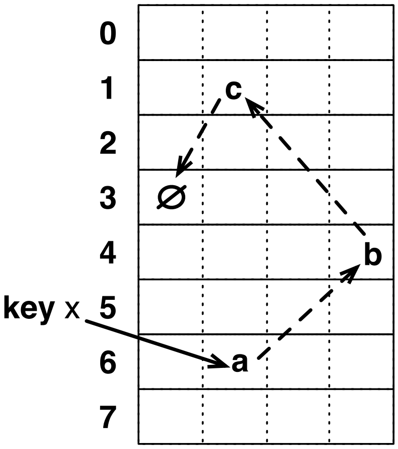


Figure 3.3: Cuckoo path. ∅ represents an empty slot.

3.3 Optimistic Cuckoo Hashing

Previous section described how to use partial-key cuckoo hashing to improve the single-thread performance; this section further extends the basic sequential cuckoo hashing algorithm to make a concurrent hash table where:

- Reader threads can concurrently access a shared hash table;
- Writer threads are serialized by a write lock;
- The on-going writer can make in-place updates on the table, concurrently with any other readers; and
- The same high table space utilization is achieved as the sequential version.

Effectively supporting concurrent access to a cuckoo hash table is challenging. The only previously proposed scheme we found improved concurrency by trading a significant amount of space [49]. Our hashing scheme is, to our knowledge, the first approach to support concurrent access (multi-reader/single-writer) while still maintaining the high space efficiency of cuckoo hashing (e.g., $> 95\%$ occupancy).

For clarity of presentation, we first define a *cuckoo path* as the sequence of displaced keys in an `Insert` operation. In Figure 3.3 "$a \Rightarrow b \Rightarrow c$" is one cuckoo path to make one bucket available to insert key x.

There are two major obstacles to making the sequential cuckoo hashing algorithm concurrent:

1. *Deadlock risk (writer/writer):* An `Insert` may modify a set of buckets when moving the keys along the cuckoo path until one key lands in an available bucket. It is not known *before swapping the keys* how many and which buckets will be modified, because each displaced key depends on the one previously kicked out. Standard techniques to make `Insert` atomic and avoid deadlock, such as acquiring all necessary locks in advance, are therefore not obviously applicable.

2. *False misses (reader/writer):* After a key is kicked out of its original bucket but before it is inserted to its alternate location, this key is unreachable from both buckets and temporarily unavailable. If `Insert` is not atomic, a reader may complete a `Lookup` and return a false miss during a key's unavailable time. E.g., in Figure 3.3, after replacing b with a at bucket 4, but before b relocates to bucket 1, b appears at neither bucket in the table. A reader looking up b at this moment may return negative results.

The only scheme previously proposed for concurrent cuckoo hashing [49] that we know of breaks up `Insert`s into a sequence of atomic displacements rather than locking the entire cuckoo path. It adds extra space at each bucket as an overflow buffer to temporarily host keys swapped from other buckets, and thus avoid kicking out existing keys. Hence, its space overhead (typically two more slots per bucket as buffer) is much higher than the basic cuckoo hashing.

Our scheme instead maintains high memory efficiency and also allows multiple-reader concurrent access to the hash table. To avoid writer/writer deadlocks, it allows only one writer at a time—a tradeoff we accept as our target workloads are read-heavy. To eliminate false misses, our design changes the order of the basic cuckoo hashing insertion by:

1) *separating discovering a valid cuckoo path from the execution of this path.* We first search for a cuckoo path, but do not move keys during this search phase.

2) *moving keys backwards along the cuckoo path.* After a valid cuckoo path is known, we first move the last key on the cuckoo path to the free slot, and then move the second to last key to the empty slot left by the previous one, and so on. As a result, each swap affects only one key at a time, which can always be successfully moved to its new location without any kickout.

Intuitively, the original `Insert` always moves a selected key to its other bucket and kicks out another existing key unless an empty slot is found in that bucket. Hence, there is always a victim key "floating" before `Insert` completes, causing false misses. In contrast, our scheme first discovers a cuckoo path to an empty slot, then propagates this empty slot towards the key for insertion along the path. To illustrate our scheme in Figure 3.3, we first find a valid cuckoo path "$a \Rightarrow b \Rightarrow c$" for key x without editing any buckets. After the path is known, c is swapped to the empty slot in bucket 3, followed by relocating b to the original slot of c in bucket 1 and so on. Finally, the original slot of a will be available and x can be directly inserted into that slot.

3.3.1 Optimization: Speeding Cuckoo Path Search

Our revised `Insert` process first looks for a valid cuckoo path before swapping the key along the path. Due to the separation of search and execution phases, we apply the following optimization to speed path discovery and increase the chance of finding an empty slot.

Instead of searching for an empty slot along a single cuckoo path, keep track of multiple paths in parallel during the `Insert` process, and terminate the searching whenever one of the paths reaches an empty slot. To be more specific, assume an `Insert` process is adding a new key x to the table. Rather than picking one victim key from x's two candidate buckets, now we select multiple keys from both buckets and make each key the head of an independent cuckoo path. These cuckoo paths are then extended in parallel and independently: at each step, multiple victim keys are "kicked out", each extending one cuckoo path; whenever one path reaches an available bucket, all the other paths are discarded and this search phase completes. In the next phase, only keys on the successful path are relocated to make room for x.

With multiple paths to search, insert may find an empty slot earlier and thus improve the throughput. In addition, it improves the chance for the hash table to store a new key before exceeding the maximum number of displacements performed, thus increasing the load factor. On the other hand, keeping track of too many cuckoo paths at the same time requires more memory than searching along a single path, and the larger memory footprint may make the searching process less CPU cache-friendly. The effect of having more cuckoo paths is evaluated in Section 3.4.

3.3.2 Optimization: Optimistic Lookup

Multiple locking schemes can work with our proposed concurrent cuckoo hashing, as long as they ensure that *during* `Insert`*, all displacements along the cuckoo path are atomic with respect to* `Lookup`*s.*

The most straightforward scheme is to associate each bucket with one lock, then acquire and release the two relevant bucket locks before and after for each displacement of `Insert` and each `Lookup`. With a careful order to avoid deadlock, this approach is simple to implement and works for workloads in general. However, this scheme requires locking twice for every `Lookup` even when writes are rare. In the following, we introduce an optimistic locking scheme that takes advantage of having a single writer to synchronize `Insert` and `Lookup`s with low overhead.

Briefly speaking, our scheme assigns a version counter for each bucket instead of a lock. The value of a version counter is only updated when a key is removed from or added to this bucket on the displacement of `Insert`; while reader threads only look for a version change during `Lookup` to detect any concurrent displacement, but never modify the version. In this way, it reduces the locking overhead for read-heavy workloads as reads are the common-case operations. To be more specific, our scheme combines two techniques: “lock striping” [49] and “optimistic locking” [56], as explained in the following.

Lock Striping It is an optimization used to store version counters. The simplest way to maintain each bucket's version is to store it inside each bucket. This approach, however, requires one counter for each bucket. Thus the storage overhead of storing all counters grows proportionally to the hash table size. For a hash table with one million buckets, it costs several megabytes to store the versions

Instead, lock striping creates an array of N counters (Figure 3.2) where N is a constant. To keep this array small, each counter is shared among multiple buckets by modulo N (e.g., the i-th counter is shared by bucket x if $x \bmod N = i$). Our implementation keeps $N = 8192$ counters in total (or 32 KB). This permits the counters to fit in CPU cache, but allows substantial concurrent access. It also keeps the chance of a “false retry” (re-reading a key due to modification of an unrelated key) very small. All counters are initialized to 0 and only read/updated by atomic memory operations to ensure the consistency among all threads.

Optimistic Locking It is the technique to coordinate readers with the single writer so these readers can detect from version change if a concurrent writer is working on the same buckets. The algorithm of optimistic locking is shown in Algorithm 1.

On the writer side, before displacing a key from bucket b_1 to b_2, an `Insert` process first increases both relevant counters by one, indicating to the other `Lookup`s an ongoing update for these two buckets; after the key is moved to its new location, both counters are again increased by one respectively to indicate the completion. As a result, the versions of both buckets are increased by two after each displacement. We will explain the reason why the version must be increased twice for each displacement later.

On the reader side, before a `Lookup` process reads the two buckets for a given key, it first snapshots the version stored in each counter: If any version is odd, there must be a concurrent `Insert` working on that bucket (or another bucket sharing the same counter), and it should wait and retry; otherwise it proceeds to the two buckets. After it finishes reading both buckets, it snapshots the counters again and compares their new versions with the old ones. If any bucket is observed to have a difference in its new and old versions, the writer must have modified this key, and the `Lookup` should retry.

It is critical to increment version twice for each displacement, because the first identifies the begin of a "critical section" and the second indicates the end. If the version counter is incremented only once (e.g. after the bucket has been successfully updated), it is possible that a reader enters the same bucket after the writer, but completes before the writer. In this case, the reader still sees the same version from its two snapshots, but it may read corrupted data due to the on-going update.

3.4 Micro-benchmark

This section investigates how the proposed techniques and optimizations contribute to performance and space efficiency.

Platform All experiments run on a machine with the following configuration. The CPU of this server is optimized for energy efficiency rather than high performance, and our system is CPU intensive, so we expect the absolute performance would be higher on "beefier" servers.

Algorithm 1: Pseudo-code for lookup and key displacement.

```
OptimisticLookup(key)
// lookup key in the hash table, return value
begin
    $b_1$, $b_2$ ← key's candidate buckets
    while true do
        $v_1$, $v_2$ ← $b_1$, $b_2$'s version (by atomic read)
        full CPU memory barrier
        if $v_1$ or $v_2$ is odd then continue
        read value of key from $b_1$ or $b_2$
        full CPU memory barrier
        $v'_1$, $v'_2$ ← $b_1$, $b_2$'s version (by atomic read)
        if $v_1 != v'_1$ or $v_2 != v'_2$ then continue
        return value
    end
end

AutomicDisplace(key, $b_1$, $b_2$)
// move key from bucket $b_1$ to bucket $b_2$
begin
    incr $b_1$ and $b_2$'s version (by atomic incr)
    full CPU memory barrier
    remove key from $b_1$
    write key to $b_2$
    full CPU memory barrier
    incr $b_1$ and $b_2$'s version (by atomic incr)
end
```

Hash table	**Size (MB)**	**# keys (million)**	**byte/key**	**Load factor**	**Constr. rate (million OPS)**	**Largest bucket**
Chaining	1280	100.66	13.33	–	14.38	13
Cuckoo 1path	1152	127.23	9.49	94.79%	6.19	4
Cuckoo 2path	1152	127.41	9.48	94.93%	7.43	4
Cuckoo 3path	1152	127.67	9.46	95.20%	7.29	4

Table 3.1: Comparison of space efficiency and construction speed of hash tables. Results in this table are independent of the key-value size. Each data point is the average of 10 runs.

CPU	2× Intel Xeon L5640 @ 2.27GHz
# cores	2×6
LLC	2×12 MB L3-cache
DRAM	2×16 GB DDR SDRAM
NIC	10Gb Ethernet

Roadmap In the following experiments, we first benchmark the construction of hash tables and measure the space efficiency. Then we examine the lookup performance of a single thread and the aggregate throughput of 6 threads all accessing the same hash table, to analyze the contribution of different optimizations. In this subsection, hash tables are linked into a workload generator directly and benchmarked on a local machine.

3.4.1 Space Efficiency and Construction Speed

We insert unique keys into empty cuckoo and chaining hash tables using a single thread, until each hash table reaches its maximum capacity. The chaining hash table, as used in Memcached, stops insertion if $1.5n$ objects are inserted to a table of n buckets to prevent imbalanced load across buckets; our cuckoo hash table stops when a single `Insert` fails to find an empty slot after 500 consecutive displacements. We initialize both types of hash tables to have a similar size (around 1.2 GB, including the space cost for pointers)

Table 3.1 shows that the cuckoo hash table is much more compact. Chaining requires 1280 MB to index 100.66 million items (i.e., 13.33 bytes per key); cuckoo hash tables are both smaller in size (1152 MB) and contain at least 20% more items, using no more than 10 bytes to index each key. Both cuckoo and chaining hash tables store only pointers to objects rather than the real key-value data; the *index* size is reduced by 1/3. A smaller index matters more for small key-value pairs.

Table 3.1 also compares cuckoo hash tables using different numbers of cuckoo paths to search for empty slots (Section 3.3.1). All of the cuckoo hash tables have high occupancy (roughly 95%). While more cuckoo paths only slightly improve the load factor, they boost construction speed non-trivially. The table with 2-way search achieves the highest construction rate (7.43 MOPS), as searching on two cuckoo paths balances the chance to find an empty slot vs. the resources required to keep track of all paths.

Chaining table construction is twice as fast as cuckoo hashing, because each insertion requires modifying only the head of the chain. Though fast, its most loaded bucket contains 13 objects in a chain (the average bucket has 1.5 objects). In contrast, bucket size in a cuckoo hash table is fixed (i.e., 4 slots), making it a better match for our targeted read-intensive workloads.

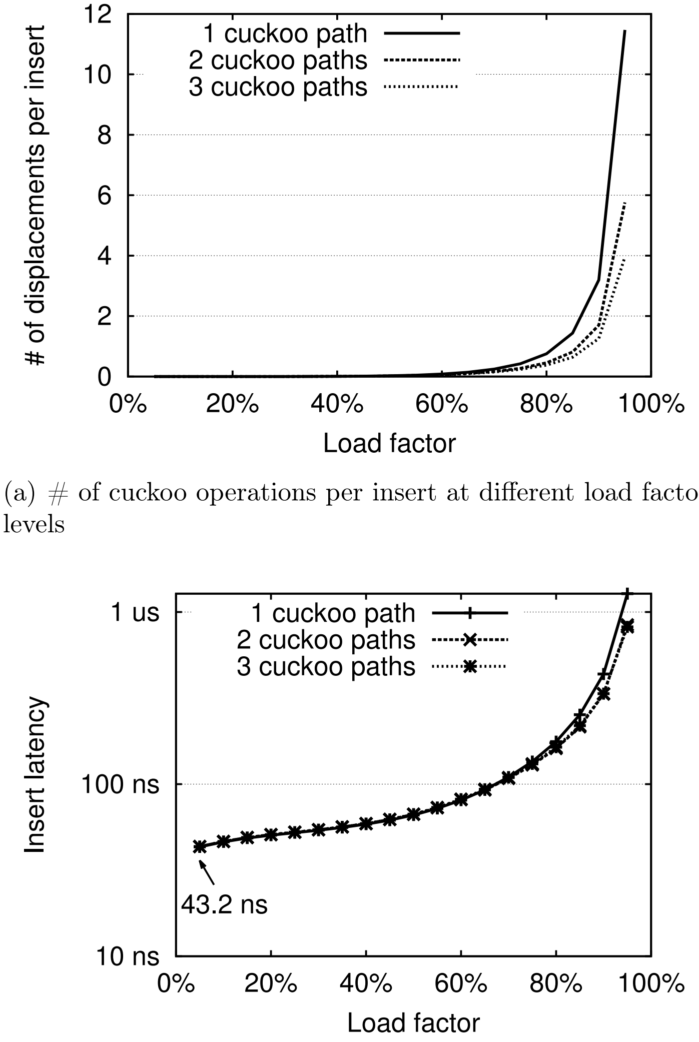
3.4.2 Insert Performance

Although the expected cost to insert one key with cuckoo hashing is O(1), it requires more displacements to find an empty slot when the table is more occupied. We therefore measure the insertion cost—in terms of both the number of displacements per insert and the latency—to a hash table with x% of all slots filled, and vary x from 0% to the maximum possible load factor. Using two cuckoo paths improves insertion latency, but using more than that has diminishing or negative returns. Figure 3.4 further shows the reciprocal throughput, expressed as latency. When the table is 70% filled, a cuckoo insert can complete within 100 ns. At 95% occupancy, insert delay is 1.3 μs with a single cuckoo path, and 0.84 μs using two.

3.4.3 Lookup Performance: Factor Analysis

This experiment investigates how much each optimization in Section 3.2 and Section 3.3 contributes to the hash table. We break down the performance gap between the basic chaining hash table used by Memcached and the final optimistic cuckoo hash table we proposed, and measure a set of hash tables—starting from the basic chaining and adding optimizations cumulatively as follows:

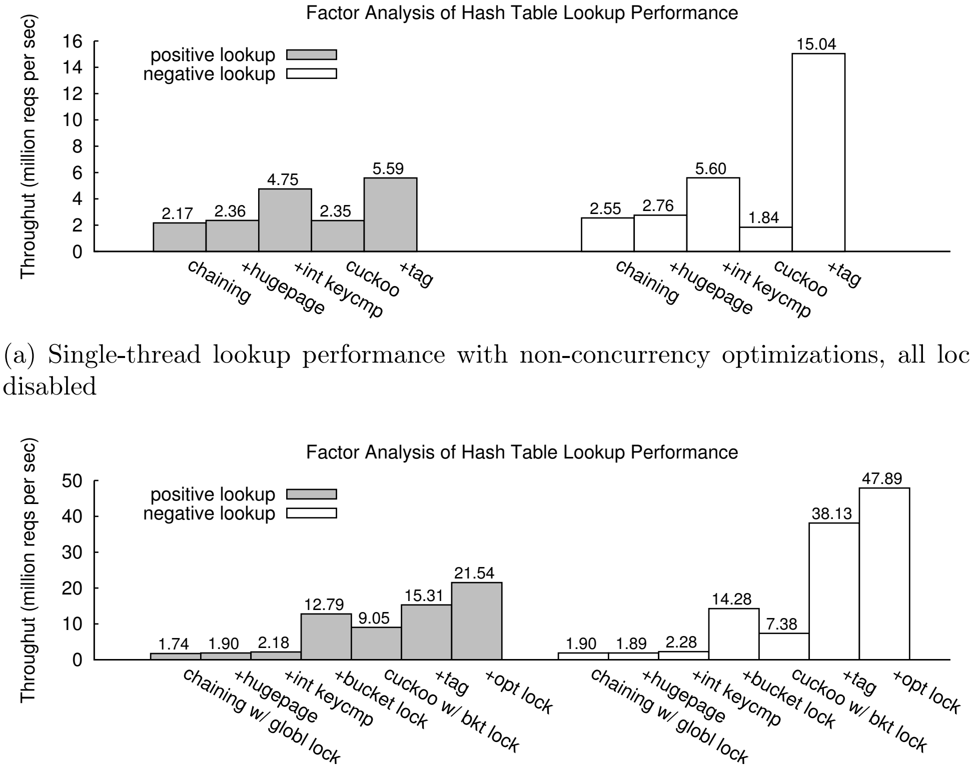
- **Chaining** is the default hash table of Memcached, serving as the baseline. A global lock is used to synchronize multiple threads.



(a) # of cuckoo operations per insert at different load factor levels

(b) Insert latency at different load factor levels

Figure 3.4: Cuckoo insert, with different number of parallel searches. Each data point is the average of 10 runs.



(a) Single-thread lookup performance with non-concurrency optimizations, all locks disabled

(b) Aggregate lookup performance of 6 threads with all optimizations

Figure 3.5: Contribution of optimizations to the hash table lookup performance. Optimizations are cumulative. Each data point is the average of 10 runs.

- **+hugepage** enables 2MB x86 hugepage support in Linux to reduce TLB misses.

- **+int keycmp** replaces the default `memcmp` (used for full key comparison) by casting each key into an integer array and then comparing based on the integers.

- **+bucket lock** replaces the global lock by bucket-based locks.

- **cuckoo** applies the naive cuckoo hashing to replace chaining, without storing the tags in buckets and using bucket-based locking to coordinate multiple threads.

- **+tag** stores the 1-byte fingerprint for each key to improve cache-locality for both `Insert` and `Lookup` (Section 3.2).

- **+opt lock** replaces the per-bucket locking scheme by optimistic locking to ensure atomic displacement (Section 3.3).

Single-thread lookup performance is shown in Figure 3.5(a) with lookups all positive or all negative. No lock is used for this experiment. In general, combining all optimizations improves performance by $\sim 2\times$ compared to the naive chaining in Memcached for positive lookups, and by $\sim 5\times$ for negative lookups. Enabling "hugepage" improves the baseline performance slightly; while "int keycmp" can almost double the performance over "hugepage" for both workloads. This is because our keys are relatively small, so the startup overhead in the built-in `memcmp` becomes relatively large. Using cuckoo hashing without the "tag" optimization *reduces* performance, because naive cuckoo hashing requires more memory references to retrieve the keys in all $4 \times 2 = 8$ candidate locations on each lookup (as described in Section 3.2). The "tag" optimization therefore significantly improves the throughput of read-only workloads (2× for positive lookups and 8× for negative lookups), because it compares the 1-byte tag first before fetching the real keys outside the table and thus eliminates a large fraction of CPU cache misses.

Multi-thread lookup performance is shown in Figure 3.5(b), measured by aggregating the throughput from 6 threads accessing the same hash table. Different from the previous experiment, a global lock is used for the baseline chaining (as in Memcached by default) and replaced by per-bucket locking and finally optimistic locking for the cuckoo hash table.

The performance gain ($\sim 12\times$ for positive and $\sim 25\times$ for negative lookups) of our proposed hashing scheme over the default Memcached hash table is large. In

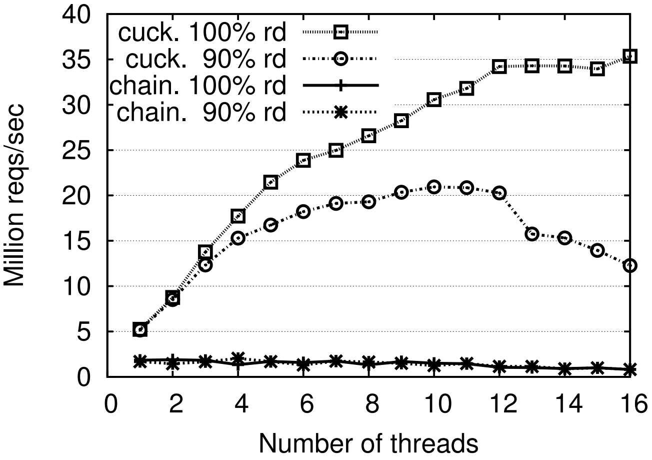


Figure 3.6: Hash table throughput vs. number of threads. Each data point is the average of 10 runs.

Memcached, all hash table operations are serialized by a global lock, thus the basic chaining hash table in fact performs worse than its single-thread throughput in Figure 3.5(a). The slight improvement ($< 40\%$) from "hugepage" and "int keycmp" indicates that most performance benefit is from making the data structures concurrent. The "bucket lock" optimization replaces the global lock in chaining hash tables and thus significantly improves the performance by 5× to 6× compared to "int keycmp". Using the basic concurrent cuckoo reduces throughput (due to unnecessary memory references), while the "tag" optimization is again essential to boost the performance of cuckoo hashing and outperform chaining with per-bucket locks. Finally, the optimistic locking scheme further improves the performance significantly.

3.4.4 Multi-core Scalability

Figure 3.6 illustrates how the total hash table throughput changes as more threads access the same hash table. We evaluate read-only and 10% write workloads. The throughput of the default hash table does not scale for either workload, because all hash table operations are serialized. Due to lock contention, the throughput is actually lower than the single-thread throughput without locks.

Using our proposed cuckoo hashing for the read-only workload, the performance scales linearly to 6 threads because each thread is pinned on a dedicated physical

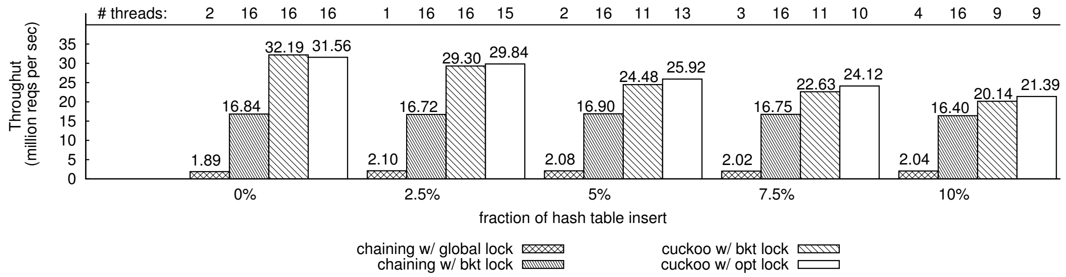


Figure 3.7: Tradeoff between lookup and insert performance. Best read + write throughput (with the number of threads needed to achieve it) is shown. Each data point is the average of 10 runs.

core on the same 6-core CPU. The next 6 threads are pinned to the other 6-core CPU in the same way. The slope of the curve becomes lower due to cross-CPU memory traffic. Threads after the first 12 are assigned to already-busy cores, and thus performance does not further increase.

With 10% `Insert`, our cuckoo hashing reaches a peak performance of 20 MOPS at 10 threads. Each `Insert` requires a lock to be serialized, and after 10 threads the lock contention becomes the bottleneck.

We further vary the fraction of insert queries in the workload and measure the best performance achieved by different hash tables. Figure 3.7 shows this best performance and also the number of threads (between 1 and 16) required to achieve this performance. In general, cuckoo hash tables outperform chaining hash tables. When more write traffic is generated, performance of cuckoo hash tables declines because `Insert`s are serialized and more `Lookup`s happen concurrently. Consequently, the best performance for 10% insert is achieved using only 9 threads; while with 100% lookup, it scales to 16 threads. Whereas the best performance of chaining hash tables (with either a global lock or per-bucket locks) keeps roughly the same when the workloads become more write-intensive.

Chapter 4

Extending Cuckoo Hashing For Cuckoo Filter

This chapter describes *cuckoo filters*, a novel data structure built upon partial-key cuckoo hashing (Chapter 3) that can replace Bloom filters [19] for many high-performance set-membership tests.

Bloom filters [19] are a widely used data structure to check whether an element x is in a set. They achieve very high space efficiency by permitting a small fraction of false positive answers (i.e., claiming that a non-existent element is in this set), but never allowing false negative answers. As a result, they are used widely for set-membership tests in many database, storage and networking systems as a critical performance optimization [23]. For example, recent popular key-value or storage systems such as HBase [3], Bigtable [24], leveldb [46], and bLSM [86] often implement a "log-structured merge tree" [77] where data changes (delete, insert, update) are appended to the end of a file to avoid in-place writes; because a point lookup might have to read from multiple files, they often rely on Bloom filters to reduce the read-amplification by preventing reads from files known not to have data.

One major limitation of Bloom filters is that they cannot delete existing items, and previous attempts to extend "standard" Bloom filters to support deletion all trade off space or performance substantially. This thesis shows that supporting deletion in approximate set-membership tests does not need to impose higher overhead in space or performance compared to standard Bloom filters. Our proposed cuckoo filter provides four major advantages in practice.

1. It supports adding and removing items dynamically;

2. It provides higher lookup performance than traditional Bloom filters, even when close to full (e.g., 95% space utilized);

3. It is much easier to implement compared to alternatives such as quotient filters; and

4. It uses even less space than Bloom filters in many practical applications, if the target false positive rate ϵ is less than 3%.

A cuckoo filter is cuckoo hash table that stores only *fingerprints*—a bit string derived from the item using a hash function—for each item inserted, instead of key-value pairs. This hash table is densely filled with fingerprints (e.g., 95% entries occupied), which confers the high space-efficiency of cuckoo filters. A set-membership query for item x simply searches the hash table for the fingerprint of x, and returns true if an identical fingerprint is found. Deleting x removes its fingerprint from the hash table.

When constructing a cuckoo filter, its fingerprint size is determined by the target false positive rate ϵ. Smaller values of ϵ require longer fingerprints to reject more false queries. An important property of cuckoo filters is a way in which they are practically better than Bloom filters for most real workloads, but are asymptotically worse: The minimum fingerprint size used in the cuckoo filter grows logarithmically with the number of entries in the table (explained in Section 4.4). As a consequence, the per-item space overhead is higher for larger tables, but this use of extra space confers a lower false positive rate. For practical problems with a few billion items or fewer, a cuckoo filter uses less space *while supporting deletion* than a non-deletable, space-optimized Bloom filter when $\epsilon < 3\%$.

The contributions of this work include

- Application of partial-key cuckoo hashing, a variant of standard cuckoo hashing to enable cuckoo filters to add and remove items dynamically (Section 4.2).

- Analysis and bounds for the false positive rate and space efficiency of cuckoo filters (Section 4.3).

- Analysis and bounds for the false positive rate and optimizations to achieve the best space efficiency (Section 4.4).

- A generalization of partial-key cuckoo hashing to support more than two hash functions per item (Section 4.5). This is challenging for cuckoo filters, because they do not store the full keys.

type	**space** **cost**	**lookup cost** **(# cache misses)**	**deletion** **support**
Bloom filter	1	k	no
blocked Bloom filter	1x	1	no
counting Bloom filter	4x	k	yes
d-left counting Bloom filter	1.5x $\sim$ 2x	d	yes
quotient filter	1x $\sim$ 1.2x	≥ 1	yes
cuckoo filter	$\leq$1x	2	yes

Table 4.1: A brief summary of Bloom filter and its variants. Assume standard and counting Bloom filters use k hash functions, and d-left counting Bloom filters have d partitions.

We also explore other advantages and limitations of cuckoo filters by comparing them with Bloom filters in Section 4.6. Finally, Section 4.7 evaluates our cuckoo filter implementation against Bloom filters and other alternatives.

4.1 Background

This section provides the background for Bloom filters (including several variants) and the challenges of cuckoo filters.

4.1.1 Bloom Filters and Variants

An approximate set-membership test returns whether an element is a member of a set, but the answer could be false with a small probability ϵ. The information theoretic optimum is essentially achievable for a *static* set by mapping each item to a fingerprint (of length $\lceil 1/\epsilon \rceil$ bits) and using a perfect hash table to store all fingerprints [23]. However, this approach is impractical because adding or removing a single item from the set requires to recompute the entire perfect hash table. To handle dynamic membership queries in general, we replace a perfect hash function with a well-designed cuckoo hash table.

In the following, we compare standard Bloom filters and the variants that include support for deletion or better lookup performance, as summarized in Table 4.1. These data structures are empirically evaluated in Section 4.7.

Standard Bloom filters [19]: They provide a compact representation of a set of items that supports two operations: `Insert` and `Lookup`. A Bloom filter allows a tunable false positive rate ϵ so that a query returns either "definitely not" (with no error), or "probably yes" (with probability ϵ of being wrong). The lower ϵ is, the more space the filter requires.

A Bloom filter consists of k hash functions and a bit array with all bits initially set to "0". To insert an item, it hashes this item to k positions in the bit array by k hash functions, and then sets all k bits to "1". Lookup is processed similarly, except it reads k corresponding bits in the array: if all the bits are set, the query returns true; otherwise it returns false. Bloom filters do not support deletion.

Bloom filters can be highly space-efficient, but are not space optimal [80]. When a false positive rate ϵ is allowed, a space-optimized Bloom filter uses

$$k = \log_2(1/\epsilon)$$

hash functions. Such a Bloom filter can store each item using

$$1.44 \log_2(1/\epsilon) \text{ bits},$$

which depends only on ϵ rather than the item size or the total number of items. In contrast, the information-theoretic minimum requires

$$\log_2(1/\epsilon) \text{ bits}$$

to store each item's membership information with a false positive rate ϵ, so a space-optimized Bloom filter imposes a 44% space overhead over the information-theoretic lower bound. It is possible for Bloom filters to retain the same false positive rate using fewer hash functions (i.e., smaller k) with a larger and sparser bit array, but then the space efficiency will be worse.

Counting Bloom filters [42]: They extend Bloom filters to allow deletions. A counting Bloom filter uses an array of counters in place of an array of bits. An insert increments the value of k counters instead of simply setting k bits, and a lookup checks if each of the required counters is non-zero. The delete operation decrements the values of these k counters. To prevent *arithmetic overflow* (i.e., incrementing a counter that has the maximum possible value), each counter in the array must be sufficiently large in order to retain the Bloom filter's properties. In practice, the counter consists of four or more bits, and a counting Bloom filter therefore requires

4× more space than a standard Bloom filter. (One can construct counting Bloom filters to use less space by introducing a secondary hash table structure to manage overflowing counters, at the expense of additional complexity.)

Blocked Bloom filters [83]: They do not support deletion, but provide better spatial locality on lookups. A blocked Bloom filter consists of an array of small Bloom filters, each fitting in one CPU cache line. Each item is stored in only one of these small Bloom filters determined by hash partitioning. As a result, every query causes at most one cache miss to load that Bloom filter, which significantly improves performance. A drawback is that the false positive rate becomes higher because of the imbalanced load across the array of small Bloom filters.

***d*-left Counting Bloom filters [21, 22]:** This data structure stores all items in d equal subtables using *d-left hashing* [67], where each item is replaced with a short fingerprint (i.e., a bit string derived from the item using a hash function). To insert an item, it selects a bucket uniformly from each subtable and places this item in the least loaded bucket. These filters delete items by removing each item's fingerprint from hash tables. Compared to counting Bloom filters, they reduce the space cost by 50%, usually requiring $1.5 - 2\times$ the space compared to a space-optimized non-deletable Bloom filter.

A *naive implementation* of d-left counting Bloom filters has a "false deletion" problem. A d-left counting Bloom filter maps each item to d candidate buckets, each from a partition of the table, and stores the fingerprint in one of its d buckets. If two different items share one and only one bucket, and they have the same fingerprint (which is likely to happen when the fingerprints are small), directly deleting the fingerprint from this specific bucket may cause a wrong item to be removed. To address this issue, d-left counting Bloom filters use an additional counter in each table slot and additional indirections [22].

Quotient filters [15]: This data structure is also built on compact hash tables to store fingerprints to support deletion. A quotient filter uses a technique similar to linear probing to locate a fingerprint, and thus provides better spatial locality. However, it requires additional meta-data to encode each entry and results in $10 \sim 25\%$ more space than a comparable standard Bloom filter. In addition, all of its operations must decode a sequence of table slots before reaching the target item; moreover, the more the hash table is filled, the longer these sequences become. As a result, its performance drops significantly when the occupancy of the hash table exceeds 75%.

In quotient filters, all fingerprints having the same quotient (i.e., the q most significant bits) must be stored in contiguous slots according to their numerical order. Thus, removing one fingerprint from a cluster of fingerprints must shift all the fingerprints after the hole by one slot, and also modify their meta-data bits [15].

4.1.2 Challenges in Cuckoo Filter

Our cuckoo filter is a cuckoo hash table that stores fingerprints of all inserted items. Storing only fingerprints creates an important challenge for dynamic item insertion, and adapting a multi-way associative hash structure creates a further challenge of needing to compare multiple potential fingerprints. Our work thus provides solutions to these problems:

- To insert new items dynamically into the cuckoo hash table, the insertion algorithm must be able to relocate existing fingerprints to their alternative locations. A space-inefficient but straightforward solution is to store each inserted item in its entirety (perhaps external to the table); given the original item ("key"), calculating its alternate location is easy. In contrast, cuckoo filters store only the fingerprints, and use *partial-key cuckoo hashing* to find an item's alternate location using only its fingerprint (Section 4.2).

- Cuckoo hashing associates each item with multiple possible locations in the hash table. This flexibility in where to store an item improves the table's occupancy, but to retain the same false positive rate when probing more possible locations on each lookup also requires longer fingerprints (leading to larger tables). In Section 4.4, we present our analysis to find the optimal table setting to balance the table occupancy and its size to minimize the average space cost per item.

4.2 Algorithms

The cuckoo filter is a cuckoo hash table that stores fingerprints for each item. Same as in chapter 3, this chapter refers to the basic unit of the hash table as a *slot* that stores one fingerprint. The hash table consists of an array of *buckets* and each bucket

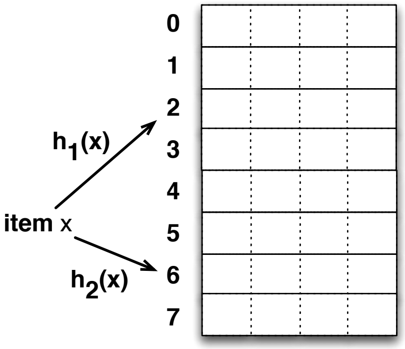


Figure 4.1: A (2,4)-cuckoo filter, two hash functions per item and four slots per bucket

has one or multiple slots. If each item has k candidate buckets and each bucket has b slots, this filter is referred to as a (k, b)-cuckoo filter in this thesis. For example, Figure 4.1 shows a (2,4)-cuckoo filter.

This section describes how cuckoo filters perform `Lookup`, `Insert` and `Delete`. Especially, the technique *partial-key cuckoo hashing* presented in Section 3.2 enables cuckoo filters to insert new items *dynamically*.

4.2.1 Insert

If we were to apply standard cuckoo hashing directly, cuckoo filters could not insert new items to an existing hash table without some external means of accessing the original items. This is because on an insertion, cuckoo hashing may relocate existing items to their alternate locations to make room for the new ones (Section 3.1). Cuckoo filters, however, only store items' fingerprints and therefore there is no way to restore and rehash the original keys to find their alternate locations (as in standard cuckoo hashing). To overcome this limitation, we apply partial-key cuckoo hashing which is the technique presented in Section 3.2 to derive an item's alternate location based on its fingerprint. As as result, to displace a key originally in bucket i, we directly calculate its alternate bucket j from the current bucket index i and the fingerprint stored in this bucket by

$$j = i \oplus \text{hash}(\text{fingerprint}). \tag{4.1}$$

Algorithm 2: `Insert(x)`

```
$f$ = fingerprint($x$);
$i_1$ = hash($x$);
$i_2 = i_1 \oplus$ hash($f$);
if bucket[$i_1$] or bucket[$i_2$] has an empty slot then
    add $f$ to that bucket;
    return Done;
end
// We must relocate existing items;
$i$ = randomly pick $i_1$ or $i_2$;
nkicks = 0;
repeat
    randomly select a slot $e$ from bucket[$i$];
    swap $f$ and the fingerprint stored in slot $e$;
    $i = i \oplus$ hash($f$);
    if bucket[$i$] has an empty slot then
        add $f$ to bucket[$i$];
        return Done;
    end
    nkicks++;
until nkicks > MaxNumKicks;
// Hashtable is considered full;
return Failure;
```

Algorithm 3: `Lookup(x)`

```
$f$ = fingerprint($x$);
$i_1$ = hash($x$);
$i_2 = i_1 \oplus$ hash($f$);
if bucket[$i_1$] or bucket[$i_2$] has $f$ then
    return True;
end
return False;
```

There are two consequences of this process that users of cuckoo hashing should note: First, while the values of h_1 and h_2 calculated by partial-key cuckoo hashing are uniformly distributed individually, they are not independent of each other (as required by standard cuckoo hashing) because of the dependence on the fingerprint. This may cause more collisions, and in particular previous analyses for cuckoo hashing (as in [37, 44]) do not hold. A full analysis of partial-key cuckoo hashing remains open; in Section 4.3, we provide a detailed discussion of this issue and consider how to achieve high occupancy for practical workloads. Second, inserting two different items x and y that have the same fingerprint is fine; it is possible to have the same fingerprint appear multiple times in a bucket. However, cuckoo filters are not suitable for applications that insert large numbers of duplicates of the same item, because the two buckets for a frequently duplicated item will quickly become overloaded. There are several solutions for such a scenario: First, if the table need not support deletion, then this issue does not arise, because only one copy of each fingerprint needs to be stored. Second, one could, at some space cost, associate counters with buckets, and increment/decrement them appropriately. Finally, if the original keys are stored somewhere (perhaps in slower external storage), one could consult that record to prevent duplicate insertion entirely, at the cost of slowing down insertion if the table already contains a (false positive) matching slot for the bucket and fingerprint. Similar requirements apply to traditional and d-left counting Bloom filters.

4.2.2 Lookup

The lookup process of a cuckoo filter is simple, as shown in Algorithm 3. Given an item x, the algorithm first calculates x's fingerprint and two candidate buckets according to Eq. (3.1). Then these two buckets are read: if any existing fingerprint in either bucket matches, the cuckoo filter returns true, otherwise the filter returns

Algorithm 4: `Delete(x)`

```
$f$ = fingerprint($x$);
$i_1$ = hash($x$);
$i_2 = i_1 \oplus$ hash($f$);
if bucket[$i_1$] or bucket[$i_2$] has $f$ then
    remove a copy of $f$ from this bucket;
    return True;
end
return False;
```

false. Notice that this ensures no false negatives as long as bucket overflow never occurs.

4.2.3 Delete

Deletion occurs by removing corresponding fingerprints from the hash tables. Other filters with similar deletion processes prove more complex than cuckoo filters. For example, as mentioned in Section 4.1.1 the d-left counting Bloom filters must use extra counters to prevent the "false deletion" problem on fingerprint collision, and quotient filters must shift a sequence of fingerprints to fill the "empty" slot after deletion and maintain their "bucket structure".

The deletion process of cuckoo filters illustrated in Algorithm 4 is much simpler. It checks both candidate buckets for a given item; if any fingerprint matches in any bucket, one copy of that matched fingerprint is then removed from that bucket.

Deletion does not have to clean the slot after deleting an item. It also avoids the "false deletion" problem when two items share one candidate bucket and also have the same fingerprint. For example, if both item x and y reside in bucket i_1 and collide on fingerprint f, partial-key cuckoo hashing ensures that they can also reside in bucket i_2 because $i_2 = i_1 \oplus \text{hash}(f)$. When deleting x, it does not matter if the process removes the copy of f added when inserting x or y. After x is deleted, the membership of y is still positive because there is one fingerprint left that must be reachable from either bucket i_1 and i_2. An important consequence of this is that the false-positive behavior of the filter is unchanged after deletion. In the above example, insertion of y into the table will cause false positives for lookups of x, by definition: they share the same bucket and fingerprint. This is the expected false-positive behavior of an approximate set membership data structure, and its probability remains bounded by ϵ.

Note that, to delete an item x safely, it must have been inserted before; otherwise, deleting a non-existent item might unintentionally remove a real, different item that happens to share the same fingerprint. This requirement also holds true for all other deletion-supporting filters.

4.3 Analysis of Asymptotic Behavior

This section shows that, with partial-key cuckoo hashing (Section 4.2), fingerprints stored in cuckoo filters must exceed a minimum size to achieve high table occupancy; this requirement grows slowly as the filter becomes larger with reasonable bucket sizes. Empirically, a filter containing up to 4 billion items can effectively fill its hash table with loads of 95% when each bucket holds four fingerprints that are 6 bits or larger.

The notation used for our analysis in this and the next section is:

ϵ	target false positive rate
f	fingerprint length in bits
α	load factor ($0 \leq \alpha \leq 1$)
b	number of slots per bucket
k	number of hash functions
m	number of buckets
n	number of items
C	average bits per item

Minimum Fingerprint Size: Our proposed partial-key cuckoo hashing derives the alternate bucket of a given item based on its current location and the fingerprint (Section 4.2). As a result, the candidate buckets for each item are not necessarily independent. Let us examine this more closely. Assume an item can be placed in bucket i_1 or i_2. According to Eq. (3.1), the number of possible values of i_2 is at most $d = 2^f$ when using f-bit fingerprints. For a table of m buckets, when $d < m$ (or equivalently $f < \log_2 m$ bits), the choice of i_2 is only a subset of all m buckets of the entire hash table. For example, using 1-byte fingerprints, given i_1 there are only up to $d = 2^f = 256$ different possible values of i_2.

Intuitively, if the fingerprints are sufficiently long, partial-key cuckoo hashing could still be a good approximation to standard cuckoo hashing; however if the hash table is very large and stores relatively short fingerprints, the chance of insertion failures will increase due to hash collisions, which may reduce the table occupancy. This situation may arise, for example, when a cuckoo filter targets a large number of items but a moderately low false positive rate. In the following, we determine analytically a lower bound of the probability of insertion failure.

Let us first derive the probability that a given set of q items collide in the same two buckets. Assume the first item x has its first bucket i_1 and a fingerprint t_x. If the other $q-1$ items have the same two buckets as this item x, they must (1) have the same fingerprint t_x, which occurs with probability $1/d$; and (2) have their first bucket either i_1 or $i_1 \oplus h(t_x)$, which occurs with probability $2/m$. Therefore the probability of such q items sharing the same two buckets is

$$\left(\frac{2}{md}\right)^{q-1}$$

Now consider a construction process that inserts n random items to an empty table of $m = cn$ buckets for a constant c and constant bucket size b. Whenever there are $q = 2b+1$ items mapped into the same two buckets, the insertion fails. This probability provides a lower bound for failure (and, we believe, dominates the failure probability of this construction process, although we do not prove this and do not need to in order to obtain a lower bound). Since there are in total $\binom{n}{2b+1}$ different possible sets of $2b+1$ items out of n items, the expected number of groups of $2b+1$ items colliding during the construction process is

$$\binom{n}{2b+1}\left(\frac{2}{md}\right)^{2b} = \binom{n}{2b+1}\left(\frac{2}{cnd}\right)^{2b} = \Omega\left(\frac{n}{d^{2b}}\right) \tag{4.2}$$

If d is $o(n^{1/2b})$, this is $\Omega(1)$. Using further calculations one can bound the probability of a failure; however, it is clear that to avoid non-trivial failures probabilities d would have to be $\Omega(n^{1/2b})$. Therefore, the fingerprint size would have to be $f = \Omega(\log n)/(2b)$ bits.

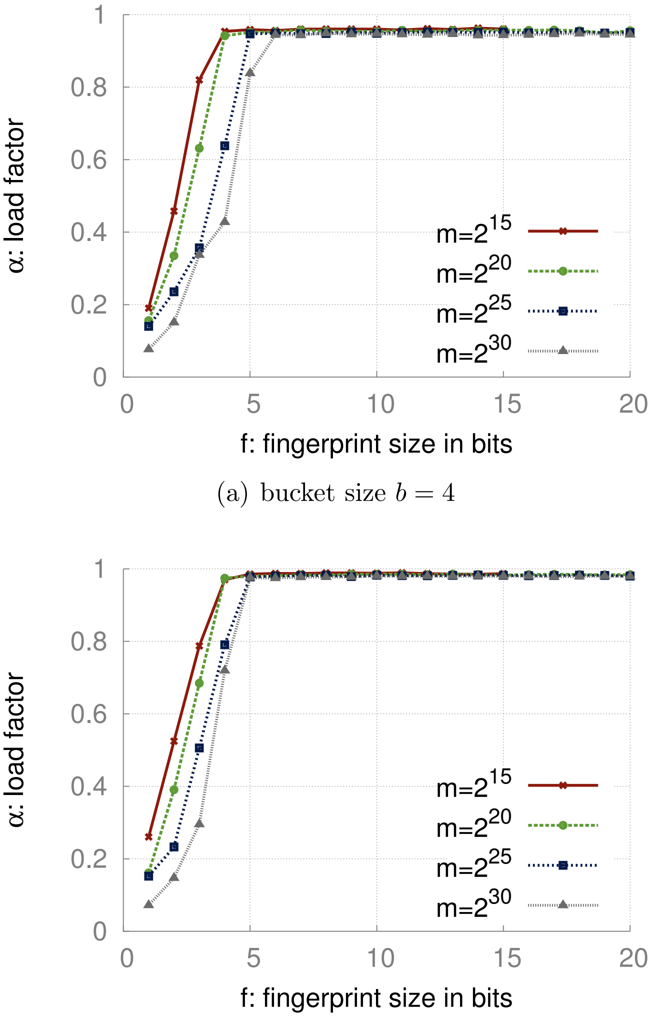
This results seems somewhat unfortunate, as the number of bits required for the fingerprint is $\Omega(\log n)$; recall that Bloom filters use a constant (approximately $ln(1/\epsilon)$ bits) per item. We might therefore have hoped that partial-key cuckoo hashing might require shorter fingerprints, and wonder about the scalability of this approach. As we show next, however, practical applications of cuckoo filters are saved by the $2b$

factor in the denominator of the lower bound: as long as we use reasonably sized buckets, the fingerprint size can remain small.

Empirical Evaluation: Figure 4.2 shows the load factor achieved with partial-key cuckoo hashing as we vary the fingerprint size f, bucket size b, and number of buckets in the table m. For the experiments, we varied the fingerprint size f from 1 to 20 bits. Random 64-bit keys are inserted to an empty filter until a single insertion relocates existing fingerprints more than 500 times before finding an empty slot (our "full" condition), then we stop and measure achieved load factor α. We run this experiment ten times for filters with $m = 2^{15}$, 2^{20}, 2^{25}, and 2^{30} buckets, and measured their minimum load over the ten trials. We did not use larger tables due to the memory constraint of our testbed machine.

As shown in Figure 4.2, across different configurations, filters with $b = 4$ could be filled to 95% occupancy, and with $b = 8$ could be filled to 98%, with sufficiently long fingerprints. After that, increasing the fingerprint size has almost no return in term of improving the load factor (but of course it reduces the false positive rate). As suggested by the theory, the minimum f required to achieve close-to-optimal occupancy increases as the filter becomes larger. Also, comparing Figure 4.2(a) and Figure 4.2(b), the minimum f for high occupancy is reduced when bucket size b becomes larger, as the theory also suggests. Overall, short fingerprints appear to suffice for realistic sized sets of items. Even when $b = 4$ and $m = 2^{30}$, so the table could contain up to four billion items, once fingerprints exceed six bits, α approaches the "optimal load factor" that is obtained in experiments using two fully independent hash functions.

Insights: The lower bound of fingerprint size derived in Eq. (4.2), together with the empirical results shown in Figure 4.2, give important insights into the cuckoo filter. While in theory the space cost of cuckoo filters is "worse" than Bloom filters—$\Omega(\log n)$ versus a constant—the constant terms are very important in this setting. For a Bloom filter, achieving $\epsilon = 1\%$ requires roughly 10 bits per item, regardless of whether one thousand, one million, or billion items are stored. In contrast, cuckoo filters require longer fingerprints to retain the same high space efficiency of their hash tables, but lower false positive rates are achieved accordingly. The $\Omega(\log n)$ bits per fingerprint, as also predicted by the theory, grows slowly if the bucket size b is sufficiently large. We find that, for practical purposes, it can be treated as a reasonable-sized constant for implementation. Figure 4.2 shows that for cuckoo



(a) bucket size $b = 4$

(b) bucket size $b = 8$

Figure 4.2: Load factor α achieved by using f-bit fingerprint using partial-key cuckoo hashing, in tables of different sizes ($m = 2^{15}, 2^{20}, 2^{25}$ and 2^{30} buckets). Short fingerprints suffice to approach the optimal load factor achieved by using two fully independent hash functions. $\alpha = 0$ means empty and 1 means completely full. Each point is the minimal load factor seen in 10 independent runs.

filters targeting a few billion items, 6-bit fingerprints are sufficient to ensure very high utilization of the hash table.

4.4 Space Optimization

The basic algorithms for cuckoo filter operations `Insert`, `Lookup`, and `Delete` presented in Section 4.2 are independent of the hash table configuration (e.g., how many slots each bucket has). However, choosing the right parameters for cuckoo filters can significantly affect space efficiency. This section focuses on optimizing the space efficiency of cuckoo filters, through parameter choices and additional mechanisms.

Another tunable parameter for the cuckoo filter is the number of hash functions. We have focused on the case of two hash functions per item because of our use of partial-key cuckoo hashing, but in Section 4.5 we discuss how to generalize partial-key cuckoo hashing to more than two hash functions.

Space efficiency is measured by the average number of bits to represent each item in a full filter, derived by the table size divided by the total number of items that a filter can effectively store. Recall that, although each slot of the hash table stores one fingerprint, not all slots are occupied: there must be some slack in the table for the cuckoo filter or there will be failures when inserting items. As a result, each item effectively costs more to store than a fingerprint: if each fingerprint is f bits and the hash table has a load factor of α, then the amortized space cost C for each item is

$$C = \frac{\text{table size}}{\#\text{ of items}} = \frac{f \cdot (\#\text{ of slots})}{\alpha \cdot (\#\text{ of slots})} = \frac{f}{\alpha} \quad \text{bits.} \tag{4.3}$$

As we will show, both f and α are related to the bucket size b. The following section studies how to minimize Eq. (4.3) given a target false positive rate ϵ by choosing the optimal bucket size b.

4.4.1 Selecting Optimal Bucket Size

Keeping a cuckoo filter's total size constant but changing the bucket size leads to two consequences:

First, larger buckets improve table occupancy (i.e., higher $b \rightarrow$ higher α). With $k = 2$ hash functions, the load factor α is 50% when bucket size $b = 1$ (i.e., the hash table is directly mapped), but increases to 84%, 95% or 98% respectively using bucket size $b = 2$, 4 or 8. Therefore, larger buckets improve table space utilization.

Second, larger buckets require longer fingerprints to retain the same false positive rate ϵ (i.e., higher $b \rightarrow$ higher f) . With larger buckets, each lookup checks more slots and thus has a higher chance to see fingerprint collisions. In the worst case of looking up a non-existent item, a query must probe two buckets where each bucket can have b slots. (While not all of these buckets may be filled, we analyze here the worst case in which they are; this gives us a reasonably accurate estimate for a table that is 95% full.) In each slot, the probability that a query is matched against one stored fingerprint and returns a false-positive successful match is *at most* $1/2^f$. After making $2b$ such comparisons, the upper bound of the total probability of a false fingerprint hit is

$$1 - (1 - 1/2^f)^{2b} \approx 2b/2^f, \tag{4.4}$$

which is proportional to the bucket size b. To retain the target false positive rate ϵ, the filter ensures $2b/2^f \leq \epsilon$, thus the minimal fingerprint size required can be approximately derived by

$$f \geq \lceil \log_2 (2b/\epsilon) \rceil = \lceil \log_2 (1/\epsilon) + \log_2 (2b) \rceil \quad \text{bits.} \tag{4.5}$$

Upper Bound of Space Cost As we have shown, both f and α depend on the bucket size b. The average space cost C by Eq. (4.3) is bound by:

$$C \leq \frac{\lceil \log_2 (1/\epsilon) + \log_2 (2b) \rceil}{\alpha}, \tag{4.6}$$

where α is also increased by b. Because b is a constant and $1/\alpha$ is 1.05 when $\alpha = 0.95$, when targeting lower false positive rate ϵ, Eq. (4.6) is asymptotically better than Bloom filters which require $1.44 \log_2 (1/\epsilon)$ bits or more for each item.

Optimal bucket size b To compare the space efficiency by using different bucket size b, we run experiments that first construct cuckoo hash tables by partial-key cuckoo hashing with different fingerprint sizes, and measure the amortized space cost per item and their achieved false positive rates. As shown in Figure 4.3, the optimal bucket size for space efficiency depends on the target false positive rate ϵ: when $\epsilon > 0.002$, having two slots per bucket yields slightly better results than using

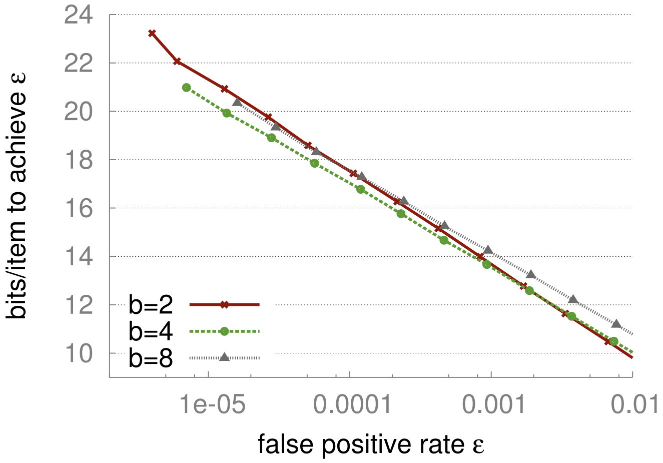


Figure 4.3: Amortized space cost per item vs. measured false positive rate, with different bucket size $b = 2, 4, 8$. Each point is the average of 10 runs

four slots per bucket; when ϵ decreases to $0.00001 < \epsilon \leq 0.002$, four slots per bucket minimizes the space cost per item.

Summary We choose $(2, 4)$-cuckoo filter (i.e., each item has two candidate buckets and each bucket has up to four fingerprints) as the default configuration, because it achieves the best or close-to-best space efficiency for the false positive rates that most practical applications [23] may be interested in. In the following, we present a technique that further saves space for cuckoo filters with $b = 4$ by encoding each bucket.

4.4.2 Semi-sorting Buckets

This subsection describes a technique for cuckoo filters with $b = 4$ slots per bucket that saves one bit per item. This optimization is based on the fact that the order of fingerprints within a bucket does not affect the query results. Based on this observation, we can compress each bucket by first sorting its fingerprints and then encoding the sequence of sorted fingerprints. This scheme is similar to the "semi-sorting buckets" optimization used in [21].

The following example illustrates how the compression saves space. Assume each bucket contains $b = 4$ fingerprints and each fingerprint is $f = 4$ bits (more general cases will be discussed later). An uncompressed bucket occupies 4 × 4 = 16 bits. However, if we sort all four 4-bit fingerprints stored in this bucket (empty slots are treated as storing fingerprints of value "0"), there are only 3876 possible outcomes in total (the number of unique combinations with replacement). If we precompute and store these 3876 possible bucket-values in an extra table, and replace the original bucket with an index into this precomputed table, then each original bucket can be represented by a 12-bit index ($2^{12} = 4096 > 3876$) rather than 16 bits, saving 1 bit per fingerprint.[1]

Note that this permutation-based encoding (i.e., indexing all possible outcomes) requires extra encoding/decoding tables and indirections on each lookup. Therefore, to achieve high lookup performance it is important to make the encoding/decoding table small enough to fit in cache. As a result, our "semi-sorting" optimization only apply this technique for tables with buckets of four slots. Also, when fingerprints are larger than four bits, only the four most significant bits of each fingerprint are encoded; the remainder are stored directly and separately.

4.5 Generalization to Use More Hash Functions

This section investigates how to generalize partial-key cuckoo hashing, so cuckoo filters could use more than two hash functions. It is a known result that the table occupancy of cuckoo hash tables could be improved when more hash functions are applied for each item [38]. For hash tables consisting of buckets having only $b = 1$ entry, when only $k = 2$ hash functions are used, the load factor is 49%; the load factor quickly improves to 91% when $k = 3$ and 97% when $k = 4$. Unfortunately, as described, the partial-key cuckoo hashing that underlies cuckoo filters can only switch between $k = 2$ possible buckets (Eq. (3.2)).

To support (k, b)-cuckoo filters when $k > 2$, one solution is to also store which hash function is used for the current location for each fingerprint. When calculating an item's alternate bucket, the insert process can pick a different hash function from the current one. This scheme is straightforward, however, it costs $\log_2 k$ extra bits per item and largely cancels the space benefit by using more hash functions.

Here we propose two schemes that add no space overhead to support more than two hash functions. The first approach is light-weight but specialized to cuckoo filters

[1] If the cuckoo filter does not need to support deletion, then it can ignore duplicate slots in the fingerprint list, creating the potential for saving an additional fraction of a bit per slot. We leave practical exploitation of this observation for future work.

			avg. # reads / lookup	
	bits per item	**load factor α**	positive query	negative query
BF	$1.44\log_2(1/\epsilon)$	–	$\log_2(1/\epsilon)$	2
CF	$(\log_2(1/\epsilon)+3)/\alpha$	95.5%	2	2
ss-CF	$(\log_2(1/\epsilon)+2)/\alpha$	95.5%	2	2

Table 4.2: Space and lookup cost of space-optimized Bloom filters (BF), cuckoo filters (CF) and cuckoo filter with semi-sorting (ss-CF).

with $k = 4$. It calculates the alternative locations by changing Eq. (3.2) to:

$$j = i \oplus \text{hash}(\text{fingerprint} + \text{a random bit}), \tag{4.7}$$

where i and j denote the current and the alternative bucket indexes. To see why this works for $k = 4$ hash functions, denote i_1=hash(item), A=hash(fingerprint + 0) and B=hash(fingerprint + 1); then, the set of all possible values of Eq. (4.7) is $\{i_1, i_1 \oplus A, i_1 \oplus B, i_1 \oplus A \oplus B\}$. Thus, when a fingerprint is kicked from its original location, it will pick a "random" one from its four candidate locations. Extending the single random bit to two bits supports $k = 16$.

The second approach supports (k, b)-cuckoo filters for general k, but is also more expensive in computation. It first splits the binary representation of v = hash(fingerprint) and i by every $\log_2 k$ bits; then it calculates the sum modulo k of each split of v and the corresponding split of i. Taking an example for v = 0111, i = 0011 and $k = 4$, v is split into 01, 11 and i turns into 00, 11. Since $(0b01 + 0b00) \bmod 4 = 0b01$ and $(0b11 + 0b11) \bmod 4 = 0b10$, the final result is hence 0110. In other words, the bit-wise xor operation for i and "hash(fingerprint)" in Eq. (3.2) is replaced with polynomial addition mod k, which will be cyclic with a period of k.

Because both schemes are either specific to certain parameters or computationally more intensive than the (2,4)-cuckoo filters, we focus on the analysis and evaluation of (2,4)-cuckoo filters in this thesis.

4.6 Comparison with Bloom Filter

We compare Bloom filters and cuckoo filters using the metrics shown in Table 4.2 and several additional factors.

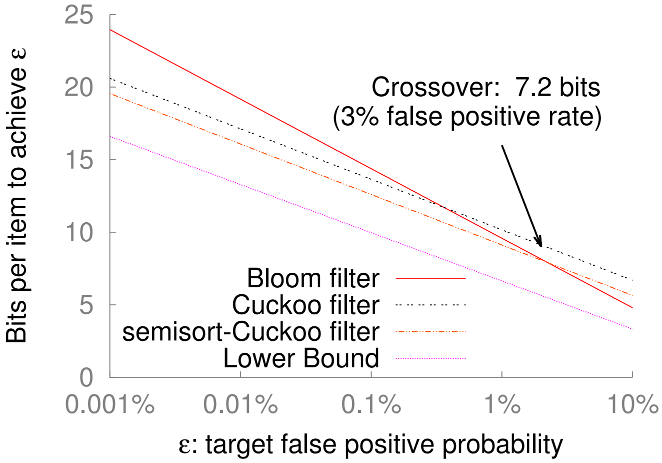


Figure 4.4: False positive rate vs. space cost per element. For low false positive rates ($< 3\%$), cuckoo filters require fewer bits per element than the space-optimized Bloom filters. The load factors to calculate space cost of cuckoo filters are obtained empirically.

Space Efficiency: Table 4.2 compares space-optimized Bloom filters and (2,4)-cuckoo filters with and without semi-sorting. Figure 4.4 further shows the bits to represent one item required by these schemes, when ϵ varies from 0.001% to 10%. The information theoretical bound requires $\log_2(1/\epsilon)$ bits for each item, and an optimal Bloom filter uses $1.44\log_2(1/\epsilon)$ bits per item, for a 44% overhead. The $(2,4)$-cuckoo filters with semi-sorting are more space efficient than Bloom filters when $\epsilon < 3\%$.

Number of Memory Accesses: For Bloom filters with k hash functions, a *positive query* must read k bits from the bit array. For space-optimized Bloom filters that require $k = \log_2(1/\epsilon)$, as ϵ gets smaller, positive queries must probe more bits and are likely to incur more cache line misses when reading each bit. For example, k equals 2 when $\epsilon = 25\%$, but the value quickly grows to 7 when $\epsilon = 1\%$, which is more commonly seen in practice. A *negative query* to a space optimized Bloom filter reads two bits on average before it returns, because half of the bits are set. Any query to a cuckoo filter, positive or negative, always reads a fixed number of buckets, resulting in (at most) two cache line misses.

Value Association: Cuckoo filters can be extended to also return an associated value (stored external to the filter) for each matched fingerprint. This property of cuckoo filters provides an approximate table lookup mechanism, which returns $1+\epsilon$ values on average for each existing item (as it can match more than one fingerprint due to false positive hits) and on average ϵ values for each non-existing item. Standard Bloom filters do not offer this functionality. Although variants like Bloomier filters can generalize Bloom filters to represent arbitrary functions instead of set membership, their algorithms are also more complex and they require more space than cuckoo filters [25].

Maximum Capacity: Cuckoo filters have a load threshold. After reaching the maximum feasible load factor, insertions are non-trivially and increasingly likely to fail, so the hash table must expand in order to store more items. In contrast, one can keep inserting new items into a Bloom filter, at the cost of an increasing false positive rate. To maintain the same target false positive rate, the Bloom filter must also expand.

Limited Duplicates: If the cuckoo filter supports deletion, it must store multiple copies of the same item. Inserting the same item $kb+1$ times will cause the insertion to fail. This is similar to counting Bloom filters where duplicate insertion causes counter overflow. However, there is no effect from inserting identical items multiple times into Bloom filters, or a non-deletable cuckoo filter.

4.7 Micro-benchmark

Implementation Our implementation[2] consists of approximately 500 lines of C++ code for standard cuckoo filters, and 500 lines for the support of the "semi-sorting" optimization presented in Section 4.4.2. In the following, we denote a basic (2,4)-cuckoo filter as "CF", and a (2,4)-cuckoo filter with semi-sorting as "ss-CF". In addition to cuckoo filters, we implemented four other filters for comparison:

- Standard Bloom filter (BF) [19]: We evaluated standard Bloom filters as the baseline. In all of our experiments, the number of hash functions k are configured to achieve the lowest false positive rate, based on the filter size and the total number of items. In addition, a performance optimization is applied to speed up lookups and inserts by doing less hashing [55]. Each insert or

[2] https://github.com/efficient/cuckoofilter

lookup only requires two hash functions $h_1(x)$ and $h_2(x)$, and then uses these two hashes to simulate the additional $k-2$ hash functions in the form of

$$g_i(x) = h_1(x) + i \cdot h_2(x).$$

- Blocked Bloom filter (blk-BF) [83]: Each filter consists of an array of blocks and each block is a small Bloom filter. The size of each block is 64 bytes to match a CPU cache line in our testbed. For each small Bloom filter, we also apply the same optimization of simulating multiple hash functions as in standard Bloom filters.

- Quotient filter (QF) [15]: We evaluated our own implementation of quotient filters[3]. This filter stores three extra bits for each item as the meta-data to help locate items. Because the performance of quotient filters degrades as they become more loaded, we set the maximum load factor to be 90% as evaluated in [15].

- d-left counting Bloom filter (dl-CBF) [21]: The filter is configured to have $d = 4$ hash tables. All hash tables have the same number of buckets; each bucket has four slots.

We emphasize that the standard and blocked Bloom filters do not support deletion, and thus are compared as a baseline.

Input Queries All the items to insert are pre-generated 64-bit integers from random number generators. We did not eliminate duplicated items because the probability of duplicates is very small.

Hash Functions: On each query, all filters first generate a 64-bit hash of the item using CityHash [1]. CityHash is a set of non-cryptographic hash functions, developed by Google to provide fast hashing from strings. Besides CityHash, we also benchmarked several other commonly used non-cryptographic hash functions including MurmurHash [71] and BobHash [20]. In general, we found that these hash functions yield very similar performance in our testbed when computing 64-bit hashes from 64-bit integers (i.e., about 22 ns per item).

[3] The source code from the original authors was not publicly available for comparison due to license issues.

metrics	CF	ss-CF	BF	blk-BF	QF	dl-CBF
# of items (M)	127.78	**128.04**	123.89	123.89	120.80	103.82
bits per item	12.60	**12.58**	13.00	13.00	13.33	15.51
false positive rate	0.19%	**0.09%**	0.19%	0.43%	0.18%	0.76%
constr. speed (MOPS)	5.00	3.13	3.91	**7.64**	1.91	4.78

Table 4.3: Space Efficiency and Construction Speed. All filters use 192MB of memory. Slots in bold are the best among the row.

The time to compute the 64-bit hash is included in our measurement. Each filter then partitions these 64 bits in this hash as it needed. For example, Bloom filters treat the high 32 bits and low 32 bits as the first two independent hashes respectively, then use these two 32-bit values to calculate the other $k-2$ hashes.

Experiment Setup: All experiments use a machine with two Intel Xeon processors (L5640 at 2.27GHz, 12MB L3 cache) and 32 GB DRAM.

Metrics: To fully understand how different filters realize the trade-offs in function, space and performance, we compare above filters by the following metrics:

- *Space efficiency*: measured by the filter size in bits divided by the number of items a full filter contains.
- *Achieved false positive rate*: measured by querying a filter with non-existing items and counting the fraction of positive return values.
- *Construction rate*: measured by the number of items that a full filter contains divided by the time to construct this full filter from empty.
- *Lookup, Insert and Delete throughput*: measured by the average number of operations a filter can perform per second. The value can depend on the workload and the occupancy of the filter.

4.7.1 Achieved False Positive Rate

We first evaluate the space efficiency and false positive rates. In each run, all filters are configured to have the same size (192 MB). Bloom filters are configured to use nine hash functions, which minimizes the false positive rate with thirteen bits per item. For cuckoo filters, their hash tables have $m = 2^{25}$ buckets each consisting of four 12-bit slots. The d-left counting Bloom filter have the same number of hash table slots, but divided into $d = 4$ partitions. Quotient filter also has 2^{27} slots where each slot stores 3-bit meta-data and a 9-bit remainder.

Each filter is initially empty and items are placed until either the filter sees an insert failure (for CF, and dl-CBF), or it has reached the target capacity limit (for BF, blk-BF, and QF). The construction rate and false positive rate of different filters are shown in Table 4.3.

Among all filters, the ss-CF achieves the lowest false positive rate. Using about the same amount of space (12.60 bits/item), enabling semi-sorting can encode one more bit into each item's fingerprint and thus halve the false positive rate from 0.19% to 0.09%, On the other hand, semi-sorting requires encoding and decoding when accessing each bucket, and thus the construction rate is reduced from 5.00 to 3.13 million items per second.

The BF and blk-BF both use 13.00 bits per item with $k = 9$ hash functions, but the false positive rate of the blocked filter is 2× higher than the BF and 4× higher than the best CF. This difference is because the blk-BF assigns each item to a single block by hashing and an imbalanced mapping will create "hot" blocks that serve more items than average and "cold" blocks that are less utilized. Unfortunately, such an imbalanced assignment happens across blocks even when strong hash functions are used [69], which increases the overall false positive rate. On the other hand, by operating in a single cache line for any query, the blk-BF achieves the highest construction rate.

The QF spends more bits per item than BFs and CFs, and achieves the second best false positive rate. Due to the cost of encoding and decoding each bucket, its construction rate is the lowest among all filters.

Finally, the dl-CBF sees insert failures and stops construction when the entire table is about 78% full, thus storing many fewer items. Its achieved false positive rate is much worse than the other filters because each lookup must check $4 \times 4 = 16$ slots, hence having a higher chance of hash collisions.

4.7.2 Lookup Performance

Different Workloads We next benchmark lookup performance after the filters are filled. This section compares the lookup throughput and latency with varying

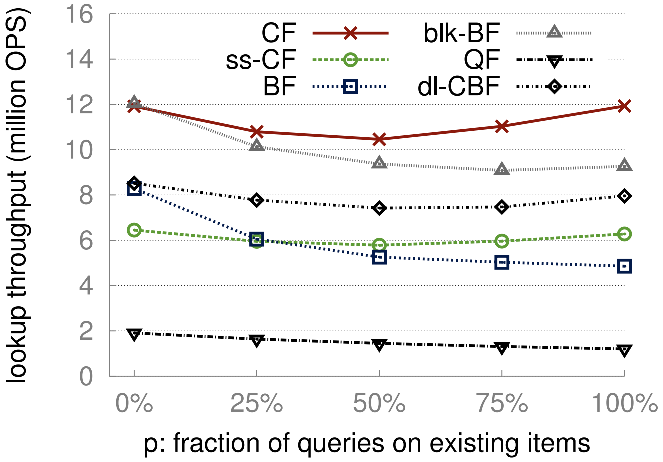


Figure 4.5: Lookup performance when a filter achieves its capacity. Each point is the average of 10 runs.

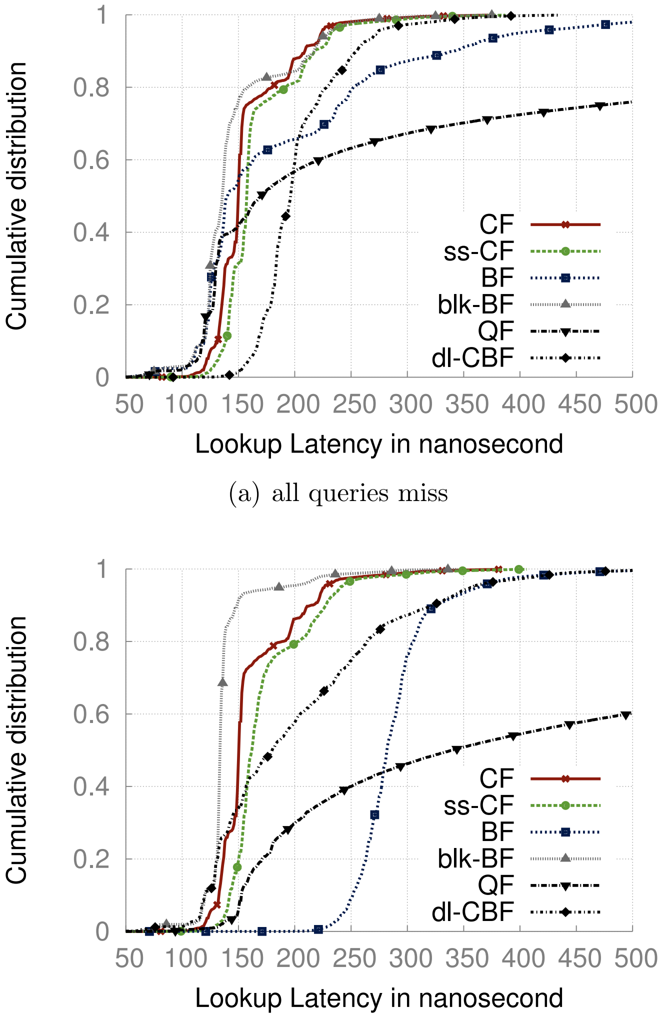
workloads. The workload is characterized by the fraction of *positive queries* (i.e., items in the table) and *negative queries* (i.e., items not in the table), which can affect the lookup speed. We vary the fraction p of positive queries in the input workload from 0% (all queries are negative) to 100% (all queries are positive).

Lookup Throughput: The benchmark result of lookup throughput is shown in Figure 4.5. Each filter occupies 192 MB, much larger than the L3 cache (20 MB) in our testbed.

The blk-BF performs well when all queries are negative, because each lookup can return immediately after fetching the first "0" bit. However, its performance declines when more queries are positive, because it must read additional bits as part of the lookup. The throughput of BF changes similarly when p increases, but is about 4 MOPS slower. This is because the BF may incur multiple cache misses to complete one lookup whereas the blocked version can always operate in one cache line and have at most one cache miss for each lookup.

In contrast, a CF always fetches two buckets[4], and thus achieves the same high performance when queries are 100% positive and 100% negative. The performance

[4] Instead of checking each item's two buckets one by one, our implementation applies a performance optimization that tries to issue two memory loads together to hide the latency of the second read.



(a) all queries miss

(b) all queries hit

Figure 4.6: CDF of lookup latency in a full filter.

drops slightly when $p = 50\%$ because the CPU's branch prediction is least accurate (the probability of matching or not matching is exactly 1/2). With semi-sorting, the throughput of CF shows a similar trend when the fraction of positive queries increases, but it is lower due to the extra decoding overhead when reading each bucket. In return for the performance penalty, the semi-sorting version reduces the false positive rate by a factor of two compared to the standard cuckoo filter. However, the ss-CF still outperforms BFs when more than 25% of queries are positive.

The QF performs the worst among all filters. When a QF is 90% filled, a lookup must search a long chain of table slots and decode each of them for the target item.

The dl-CBF outperforms the ss-CF, but is about 30% slower than a BF. It also keeps about the same performance when serving all negative queries and all positive queries, because only a constant number of slots are searched on each lookup.

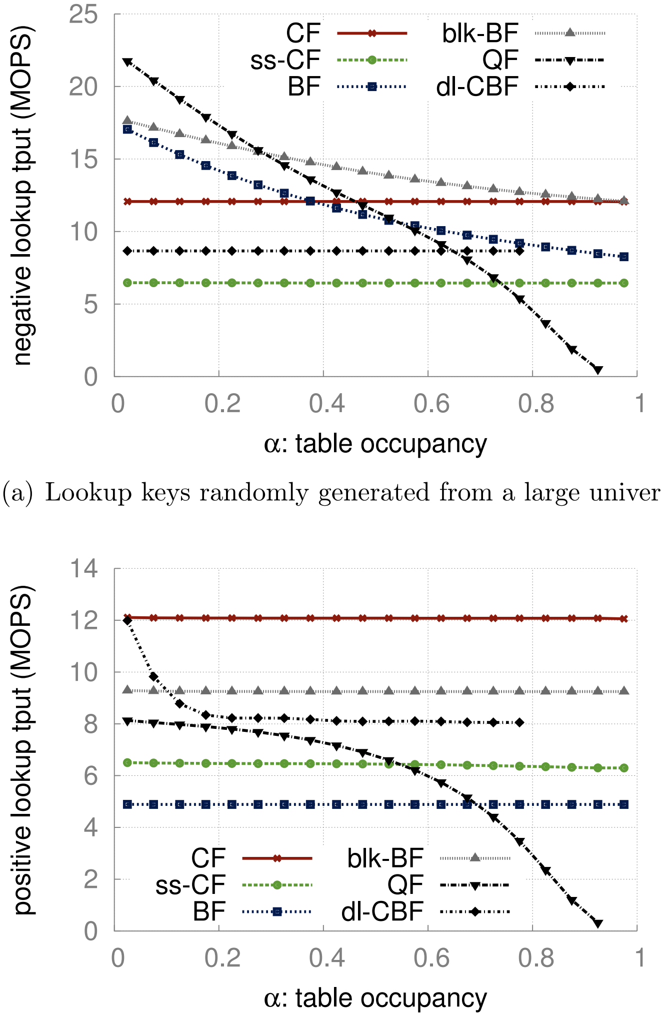
Lookup Latency: We also examine the lookup latency of different filters and show their cumulative distribution in Figure 4.6.

The CF and blk-BF have lower latency, because each lookup costs at most one cache miss for blk-BF and two for CF. So most of their lookups complete in 250 ns. The BF and dl-CBF have longer tails and complete most lookups in 400 ns, because they may need more memory references on one lookup. The QF, because it searches a large and non-constant number of slots in a row for a target item, has a long tail.

One interesting observation is that the latency is not entirely the inverse of throughput. In fact, according to Figure 4.5 the blk-BF has lower throughput than CF, whereas Figure 4.6 shows that its latency is the lowest. We suspect this occurs because the cuckoo filter has a shorter code path and thus it benefits more from hardware-level parallelism such as multi-issue and out-of-order execution.

The workload also affects the latency distribution. Comparing Figure 4.6(a) and Figure 4.6(b), the tails of BF, dl-CBF and QF become longer for positive queries because they must access more memory, while the CF and blk-BF stay the same.

Different Occupancy In this experiment, we measure the lookup throughput when the these filters are filled at different levels of occupancy. We vary their load factor α of each filter from 0 (empty) to its maximum occupancy. Figure 4.7 shows the average instantaneous lookup throughput when all queries are negative (i.e., for non-existing items) and all queries are positive (i.e., for existing items).



(a) Lookup keys randomly generated from a large universe

(b) Lookup keys uniformly selected from keys existing in the filter

Figure 4.7: Lookup throughput (MOPS) at different occupancy. Each point is the average of 10 runs.

The throughput of both CF and ss-CF is mostly stable across different load factor levels, for both negative and positive queries. This is because the total number of slots to read and compare remains constant even as more items are inserted.

In contrast, the throughput of QF decreases substantially as the filter is more loaded. This filter has to search an increasingly long chain of slots for the target item as the load factor grows.

Both BF and blk-BF behave differently when serving negative and positive queries. For positive queries, they must always check in total k bits, no matter how many items have been inserted, thus providing constant lookup throughput; while for negative queries, when the filter is less loaded, fewer bits are set and a lookup can return earlier when seeing a "0".

The dl-CBF behaves differently from the BF. When all lookups are negative, it ensures constant throughput like the CF, because a total of 16 slots from four buckets must be searched, no matter how many items this filter contains. For positive queries, if there are fewer items inserted, the lookup may return earlier before all four buckets are checked; however, this difference becomes negligible after the dl-CBF is about 20% filled.

4.7.3 Insert Performance

The overall construction speed, measured based on the total number of items a full filter contains and the total time to insert these items, is shown in Table 4.3. We also study the instantaneous insert throughput across the construction process. Namely, we measure the insert throughput of different filters when they are at levels of load factors, as shown in Figure 4.8.

In contrast to the lookup throughput shown in Figure 4.7, both types of CF have decreasing insert throughput when they are more filled (though their overall construction speed is still high), while both BF and blk-BF ensure almost constant insert throughput. The CF may have to move a sequence of existing keys recursively before successfully inserting a new item, and this process becomes more expensive when the load factor grows higher. In contrast, both Bloom filters always set k bits, regardless of the load factor.

The QF also has decreasing insert throughput. This is because it must shift a sequence of items before inserting a new item, and this sequence also grows longer when the table is more filled.

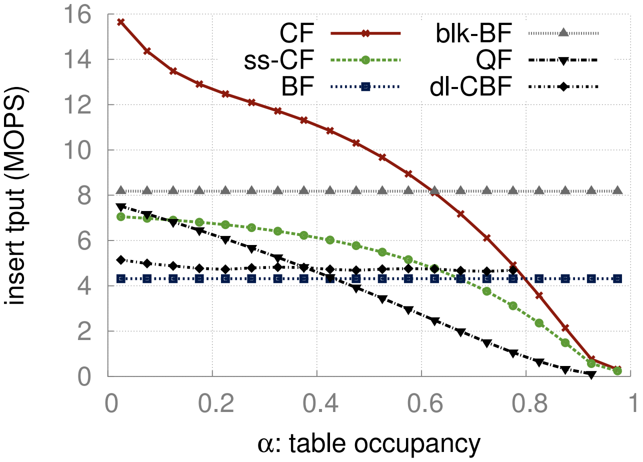


Figure 4.8: Insert throughput (MOPS) at different occupancy. Insert keys randomly generated from a large universe until each data structure achieves its designed capacity. Each point is the average of 10 runs.

The dl-CBF keeps constant throughput. For each insert, it must only find an empty slot in up to four buckets; once such an slot cannot be found, the insert stops the construction process without relocating existing items as in cuckoo hashing. This is also why its maximum load factor is no more than 80%.

4.7.4 Delete Performance

Figure 4.9 shows the delete performance of filters that support deletion. The experiment deletes keys from an initially full filter until it is empty. The CF achieves the highest throughput. Both CF and ss-CF provide stable performance through the entire process. The dl-CBF performs the second best among all filters. The QF is the slowest when close to full, but becomes faster than ss-CF when close to empty.

4.7.5 Evaluation Summary

The CF ensure high and stable lookup performance for different workloads and at different levels of occupancy. Its insert throughput declines as the filter is more filled, but the overall construction rate is still faster than other filters except the blk-BF. Enabling semi-sorting makes cuckoo filters more space-efficient than space-optimized

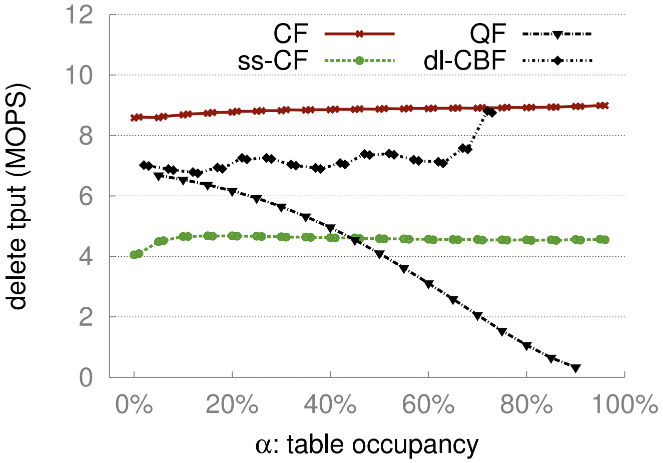


Figure 4.9: Delete-until-empty throughput (MOPS) at different occupancy. Each point is the average of 10 runs.

Bloom filters. It also makes lookups, inserts, and deletes slower, but still faster than conventional Bloom filters.

The blk-BF performs much better than the standard BF; however it does not support delete and its false positive rate is also much higher at the same space cost. The QF is in general slower than the other filters, and its insert/lookup performance declines significantly when the occupancy becomes higher. The dl-CBF achieves the worst space efficiency.

4.8 Related Work

Prior efforts that extend Bloom filters for deletion, such as classic counting Bloom filters [42], *d*-left counting Bloom filters [21, 22], and quotient filters [15] are described in Section 4.1. Cuckoo filters achieve higher space efficiency and performance than these data structures.

Other previous work has proposed improvements to Bloom filters, either in space efficiency and/or in performance. Rank-Indexed Hashing [51] is similar to the *d*-left counting Bloom filter, but builds linear chaining hash tables to store compressed fingerprints. Although it achieves higher space efficiency than *d*-left counting Bloom filters, updating the internal index that reduces the chaining cost is very expensive, making it less appealing in dynamic settings.

Putze et al. proposed two variants of Bloom filters [83]. The first, called a *Block Bloom filter*, builds on earlier work by Manber and Wu [62]. This filter is described in Section 4.1 and evaluated in Section 4.7. The second variant, called a *Golomb-Compressed Sequence* stores all items' fingerprints in a sorted list. Its space is near-optimal, but the data structure is static and requires non-constant lookup time to decode the encoded sequence. Thus it is not evaluated with other filters in this chapter.

Pagh et al. proposed an asymptotically space-optimal hash data structure [80] based on Cleary [27]. This data structure, however, is substantially more complex than its alternatives and does not yet appear amenable to a high performance implementation. In contrast, cuckoo filters are very easy to implement.

Chapter 5

Efficient Key-Value Stores

The load-balanced key-value system presented in Chapter 2 requires two core components: a memory-speed key-value cache placed at the frontend and a cluster of persistent key-value stores to form the backend. This chapter investigates the design and implementation of efficient key-value stores meeting this requirement:

- Section 5.1 presents MemC3 [41], a key-value cache that serves data from memory and thus provides low-latency retrieval with scalable performance on multi-core CPUs.

- Section 5.2 introduces SILT [59], a high-performance key-value store that effectively uses SSDs to serve key-value data at extremely low memory overhead.

The design of these two types of key-value stores exploits:

- advanced/space-efficient data structures, e.g., optimistic cuckoo hashing in Chapter 3 and the cuckoo filter in Chapter 4; and

- architecture-aware optimizations, e.g., exploiting CPU cache locality to minimize the number of memory fetches required to complete one query; or exploiting fast sequential writes and fast random reads on flash while avoiding small random writes.

function	stock Memcached	MemC3
Hash Table		
concurrency	serialized	concurrent lookup, serialized insert
lookup performance	slower	faster
insert performance	faster	slower
space	13.3n bytes	$\sim 9.7n$ bytes
Cache Mgmt		
concurrency	serialized	concurrent update, serialized eviction
space	18n bytes	n bits

Table 5.1: Comparison of operations. n is the number of existing key-value items.

5.1 MemC3: In-Memory Key-Value Cache

Low-latency access to data has become critical for many Internet services in recent years. This requirement has led many system designers to serve all or most of certain data sets from main memory—using the memory either as their primary store [65, 70, 85, 88] or as a cache to deflect hot or particularly latency-sensitive items [39].

This section demonstrates that careful attention to algorithm and data structure design can significantly improve throughput and memory efficiency for in-memory data stores. We show that traditional approaches often fail to leverage the target system's architecture and expected workload. As a case study, we focus on Memcached [65], a popular in-memory caching layer, and show how our toolbox of techniques can improve Memcached's performance by 3× and reduce its memory use by 30%.

We implement and evaluate MemC3, a networked, in-memory key-value cache, based on Memcached-1.4.13. Our prototype supports the most important Memcached commands including `GET`, `SET`, `ADD`, `REPLACE`, `DELETE`. [1]

Table 5.1 compares MemC3 and stock Memcached. MemC3 provides higher throughput using significantly less memory and computation as we will demonstrate in the remainder of this section.

[1] It does not yet provide the full memcached API with missing commands like `INCR`, `DECR` and `CAS`.

5.1.1 Background: Memcached Overview

Memcached is an in-memory key-value store, designed for serving small pieces of arbitrary data without reading disks. It is often used to cache strings or objects, such as the results of database queries, API calls, or page rendering. Since all data is only stored and served from memory, Memcached does not guarantee its durability. To recover data from system crashes, Memcached servers are often backed by non-volatile storage such as database servers.

Interface Memcached implements a simple and light-weight key-value interface where all key-value tuples are stored in and served from DRAM. Clients communicate with the Memcached servers over the network using the following commands:

- `SET/ADD/REPLACE(key, value)`: add a (key, value) object to the cache;
- `GET(key)`: retrieve the value associated with a key;
- `DELETE(key)`: delete a key.

Internally, Memcached uses a hash table to index the key-value entries. These entries are also in a linked list sorted by their most recent access time. The least recently used (LRU) entry is evicted and replaced by a newly inserted entry when the cache is full.

Hash Table To lookup keys quickly, the location of each key-value entry is stored in a hash table. Hash collisions are resolved by chaining: if more than one key maps into the same hash table bucket, they form a linked list. Chaining is efficient for inserting or deleting single keys. However, lookup may require scanning the entire chain.

Memory Allocation Naive memory allocation (e.g., malloc/free) could result in significant memory fragmentation. To address this problem, Memcached uses *slab-based memory allocation*. Memory is divided into 1 MB pages, and each page is further sub-divided into fixed-length *chunks*. Key-value objects are stored in an appropriately-size chunk. The size of a chunk, and thus the number of chunks per page, depends on the particular slab class. For example, by default the chunk size of slab class 1 is 72 bytes and each page of this class has 14563 chunks; while the chunk size of slab class 43 is 1 MB and thus there is only 1 chunk spanning the whole page.

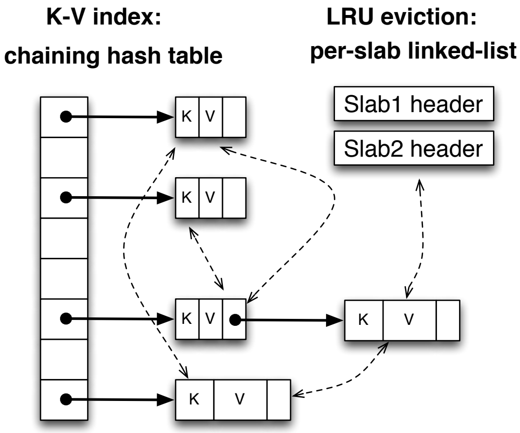


Figure 5.1: Memcached data structures.

To insert a new key, Memcached looks up the slab class whose chunk size best fits this key-value object. If a vacant chunk is available, it is assigned to this item; if the search fails, Memcached will execute cache eviction.

Cache policy In Memcached, each slab class maintains its own objects in an LRU queue (see Figure 5.1). Each access to an object causes that object to move to the head of the queue. Thus, when Memcached needs to evict an object from the cache, it can find the least recently used object at the tail. The queue is implemented as a doubly-linked list, so each object has two pointers.

Threading Memcached was originally single-threaded. It uses libevent for asynchronous network I/O callbacks [82]. Later versions support multi-threading but use global locks to protect the core data structures. As a result, operations such as index lookup/update and cache eviction/update are all serialized. Previous work has shown that this locking prevents current Memcached from scaling up on multi-core CPUs [48].

Performance Enhancement Previous solutions [16, 50, 66] shard the in-memory data to different cores. Sharding eliminates the inter-thread synchronization to permit higher concurrency, but under skewed workloads it may also exhibit imbalanced load across different cores or waste the (expensive) memory capacity. Instead of simply sharding, we explore how to scale performance to many threads that share

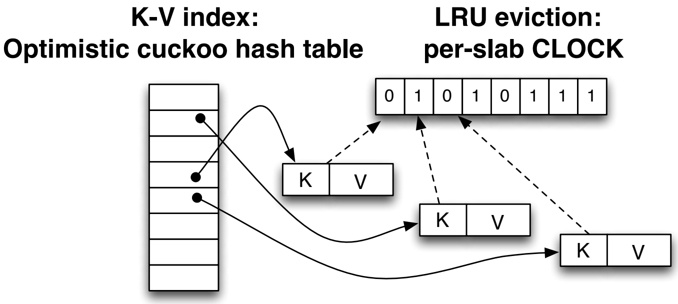


Figure 5.2: Overview of MemC3: optimistic cuckoo hashing and CLOCK LRU.

and access the same memory space; one could then apply sharding to further scale the system.

5.1.2 Design

MemC3 applies the optimistic cuckoo hashing proposed in Chapter 3 to index all key-value pairs, as shown in Figure 5.2. As a result, the key-value index achieves over 95% space occupancy and allows concurrent read access without locking. Besides, Cache management and eviction is also important. The strict LRU cache management is also a synchronization bottleneck, as all updates to the cache must be serialized in Memcached. To make the cache management *space efficient* and *concurrent* (no synchronization to update LRU), we implement an *approximate LRU cache* based on the CLOCK replacement algorithm [29]. CLOCK is a well-known algorithm; our contribution lies in integrating CLOCK replacement with the optimistic, striped locking in our cuckoo algorithm to reduce both locking and space overhead.

As our target workloads are dominated by small objects, the space saved by trading perfect for approximate LRU allows the cache to store significantly more entries, which in turn improves the hit ratio. As we will show in Section 5.1.3, our cache management achieves 3× to 10× the query throughput of the default cache in Memcached, while also improving the hit ratio.

CLOCK Replacement to Approximate LRU A cache must implement two functions related to its replacement policy:

- `Update` to keep track of the recency after querying a key in the cache; and

- Evict to select keys to purge when inserting keys into a full cache.

Memcached keeps each key-value entry in a doubly-linked-list based LRU queue within its own slab class. After each cache query, Update moves the accessed entry to the *head* of its own queue; to free space when the cache is full, Evict replaces the entry on the *tail* of the queue by the new key-value pair. This ensures strict LRU eviction in each queue, but unfortunately it also requires two pointers per key for the doubly-linked list and, more importantly, all Updates to one linked list are serialized. Every read access requires an update, and thus the queue permits no concurrency even for read-only workloads.

CLOCK approximates LRU with improved concurrency and space efficiency. For each slab class, we maintain a *circular buffer* and a *virtual hand*; each bit in the buffer represents the recency of a different key-value object: 1 for "recently used" and 0 otherwise. Each Update simply sets the recency bit to 1 on each key access; each Evict checks the bit currently pointed by the hand. If the current bit is 0, Evict selects the corresponding key-value object; otherwise we reset this bit to 0 and advance the hand in the circular buffer until we see a bit of 0.

Integration with Optimistic Cuckoo Hashing The Evict process must coordinate with reader threads to ensure the eviction is safe. Otherwise, a key-value entry may be overwritten by a new (key,value) pair after eviction, but threads still accessing the entry for the evicted key may read dirty data. To this end, the original Memcached adds to each entry a 2-byte reference counter to avoid this rare case. Reading this per-entry counter, the Evict process knows how many other threads are accessing this entry concurrently and avoids evicting those busy entries.

Our cache integrates cache eviction with our optimistic locking scheme for cuckoo hashing. When Evict selects a victim key x by CLOCK, it first increases key x's version counter to inform other threads currently reading x to retry; it then deletes x from the hash table to make x unreachable for later readers, including those retries; and finally it increases key x's version counter again to complete the change for x. Note that Evict and the hash table Insert are mutually serialized using a common lock, so when updating the counters they cannot affect each other.

With Evict as above, our cache ensures consistent GETs by version checking. Each GET first snapshots the version of the key before accessing the hash table; if the hash table returns a valid pointer, it follows the pointer and reads the value associated. Afterwards, GET compares the latest key version with the snapshot. If the versions

Algorithm 5: Psuedo code of `GET` in MemC3

```
GET(key)   //get value of key from cache
begin
    while true do
        vs = ReadCounter(key);   //key version
        ptr= Lookup(key);        //check hash table
        if ptr == NULL then  return NULL ;
        prepare response for data in ptr;
        ve = ReadCounter(key);   //key version
        if vs & 1 or vs != ve then
            //may read dirty data, try again
            continue
        end
        Update(key);             //update CLOCK
        return response
    end
end
```

differ, then `GET` may have observed an inconsistent intermediate state and must retry. The pseudo-code of `SET` and `GET` is shown in Algorithm 6 and Algorithm 5.

5.1.3 Evaluation

Workload We use YCSB [28] to generate 100 million key-value queries, following a zipf distribution. Each key is 16 bytes and each value 32 bytes. We evaluate caches with four configurations:

- **chaining+LRU**: the default Memcached cache configuration, using chaining hash table to index keys and LRU for replacement;
- **cuckoo+LRU**: keeping LRU, but replacing the hash table by concurrent optimistic cuckoo hashing with all optimizations proposed;
- **chaining+CLOCK**: an alternative baseline combining optimized chaining with the CLOCK replacement algorithm. Because CLOCK requires no serialization to update, we also replace the global locking in the chaining hash table with the per-bucket locks; we further include our engineering optimizations such as "hugepage", "int keycmp".

Algorithm 6: Psuedo code of SET in MemC3

```
SET(key, value)    //insert (key,value) to cache
begin
    lock();
    ptr = Alloc();        //try to allocate space
    if ptr == NULL then
        ptr = Evict();    //cache is full, evict old item
    end
    memcpy key, value to ptr;
    Insert(key, ptr); //index this key in hashtable
    unlock();
end
```

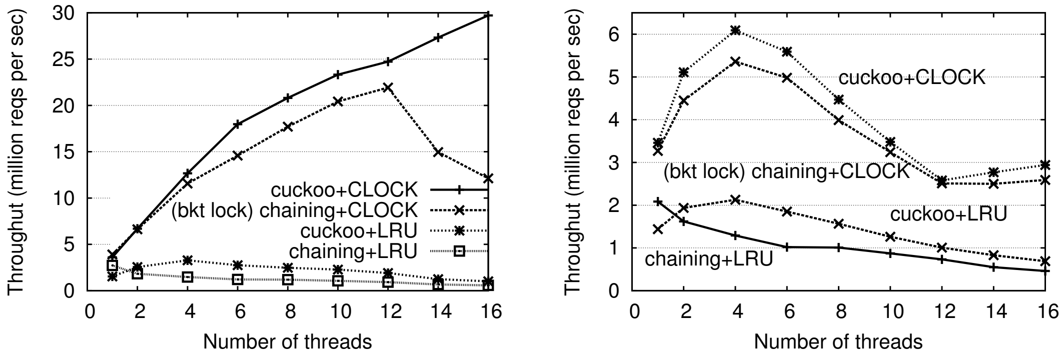
- **cuckoo+CLOCK**: the data structure of MemC3, using cuckoo hashing to index keys and CLOCK for replacement.

We vary the cache size from 64 MB to 10 GB. Note that this cache size parameter does not count the space for the hash table, only the space used to store key-value objects. All four types of caches are linked into a workload generator and micro-benchmarked locally.

Cache Throughput Because each GET miss is followed by a SET to the cache, to understand the cache performance with heavier or lighter insertion load, we evaluate two settings:

- a read-only workload on a "big" cache (i.e., 10 GB, which is larger than the working set), which had no cache misses or inserts and is the best case for performance;
- a write-intensive workload on a "small" cache (i.e., 1 GB, which is ∼10% of the total working set) where about 15% GETs miss the cache. Since each miss triggers a SET in turn, a workload with 15% inserts is worse than the typical real-world workload reported by Facebook [12].

Figure 5.3(a) shows the results of benchmarking the "big cache". Though there are no inserts, the throughput does not scale for the default cache (chaining+LRU), due to lock contention on each LRU update (moving an object to the head of the linked



(a) 10GB "big cache" (> working set): 100% GETs hit

(b) 1GB "small cache" (< working set): 85% GETs hit

Figure 5.3: Cache throughput vs. number of threads. Each data point is the average of 10 runs.

list). Replacing default chaining with the concurrent cuckoo hash table improves the peak throughput slightly. This suggests that only having a concurrent hash table is not enough for high performance. After replacing the global lock with bucket-based locks and removing the LRU synchronization bottleneck by using CLOCK, the chaining-based cache achieves 22 MOPS at 12 threads, and drops quickly due to the CPU overhead for lock contention after all 12 physical cores are assigned. Our proposed cuckoo hash table combined with CLOCK, however, scales to 30 MOPS at 16 threads.

Figure 5.3(b) shows that peak performance is achieved at 6 MOPS for the "small cache" by combining CLOCK and cuckoo hashing. The throughput drop is because the 15% GET misses result in about 15% hash table inserts, so throughput drops after 6 threads due to serialized inserts.

Space Efficiency Table 5.2 compares the maximum number of items (16-byte key and 32-byte value) a cache can store given different cache sizes[2]. The default LRU with chaining is the least memory efficient scheme. Replacing chaining with cuckoo hashing improves the space utilization slightly (7%), because one pointer (for hash table chaining) is eliminated from each key-value object. Keeping chaining but

[2] The space to store the index hash tables is separate from the given cache space in Table 5.2. We set the hash table capacity larger than the maximum number of items that the cache space can possibly allocate. If chaining is used, the chaining pointers (inside each key-value object) are also allocated from the cache space.

	cache size					
cache type	64 MB	128 MB	256 MB	512 MB	1 GB	2 GB
	# items stored (million)					
chaining+LRU	0.60	1.20	2.40	4.79	9.59	19.17
cuckoo+LRU	0.65	1.29	2.58	5.16	10.32	20.65
chaining+CLOCK	0.76	1.53	3.05	6.10	12.20	24.41
cuckoo+CLOCK	**0.84**	**1.68**	**3.35**	**6.71**	**13.42**	**26.84**
	cache miss ratio with 95% `GET`, 5% `SET`, zipf distribution					
chaining+LRU	36.34%	31.91%	27.27%	22.30%	16.80%	10.44%
cuckoo+LRU	**35.87%**	**31.42%**	**26.76%**	21.74%	16.16%	9.80%
chaining+CLOCK	37.07%	32.51%	27.63%	22.20%	15.96%	8.54%
cuckoo+CLOCK	36.46%	31.86%	26.92%	**21.38%**	**14.68%**	**7.89%**

Table 5.2: Comparison of four types of caches. Results in this table depend on the object size (16-byte key and 32-byte value used). Bold entries are the best in their columns. Each data point is the average of 10 runs.

replacing LRU with CLOCK improves space efficiency by 27% because two pointers (for LRU) and one reference count are saved per object. Combining CLOCK with cuckoo increases the space efficiency by 40% over the default. The space benefit arises from eliminating three pointers and one reference count per object.

Cache Miss Ratio Compared to the linked list based approach in Memcached, CLOCK approximates LRU eviction with much lower space overhead. This experiment sends 100 million queries (95% `GET` and 5% `SET`, in zipf distribution) to a cache with different configurations, and measures the resulting cache miss ratios. Note that each `GET` miss will trigger a retrieval to the backend database system, therefore reducing the cache miss ratio from 10% to 7% means a reduction of traffic to the backend by 30%. Table 5.2 shows when the cache size is smaller than 256 MB, the LRU-based cache provides a lower miss ratio than CLOCK. LRU with cuckoo hashing improves upon LRU with chaining, because it can store more items. In this experiment, 256 MB is only about 2.6% of the 10 GB working set. Therefore, when the cache size is very small, CLOCK—which is an approximation—has a higher chance of evicting popular items than strict LRU. For larger caches, CLOCK with cuckoo hashing outperforms the other two schemes because the extra space improves the hit ratio more than the loss of precision decreases it.

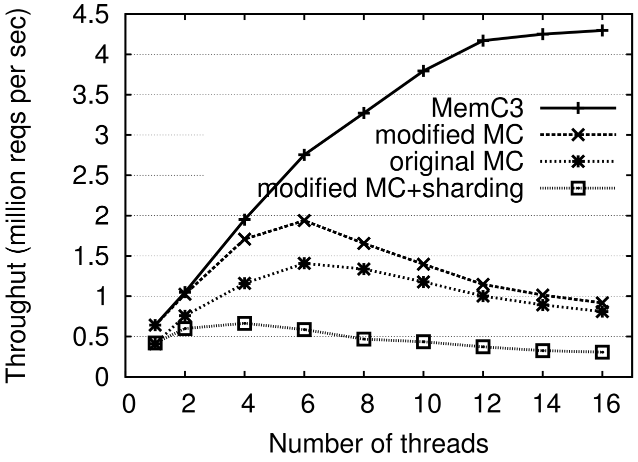


Figure 5.4: Full system throughput (over network) v.s. number of server threads

Full System Throughput This experiment uses the same workload as in previous experiment, with 95% GETs and 5% SETs generated by YCSB with zipf distribution. MemC3 runs on the same server as before, but the clients are 50 different nodes connected by a 10GB Ethernet. The clients use libmemcached 1.0.7 [58] to communicate with our MemC3 server over the network. To amortize the network overhead, we use multi-get supported by libmemcached [58] by batching 100 GETs.

In this experiment, we compare four different systems: original Memcached, optimized Memcached (with non-algorithmic optimizations such as "hugepage", "in keycmp" and tuned CPU affinity), optimized Memcached with sharding (one core per Memcached instance) and MemC3 with all optimizations enabled. Each system is allocated with 1GB memory space (not including hash table space).

Figure 5.4 shows the throughput as more server threads are used. Overall, the maximum throughput of MemC3 (4.4 MOPS) is almost 3× that of the original Memcached (1.5 MOPS). The non-algorithmic optimizations improve throughput, but their contribution is dwarfed by the algorithmic and data structure-based improvements.

A surprising result is that today's popular technique, *sharding, performs the worst* in this experiment. This occurs because the workload generated by YCSB is heavy-tailed, and therefore imposes differing load on the memcached instances. Those serving "hot" keys are heavily loaded while the others are comparatively idle. While the severity of this effect depends heavily upon the workload distribution, it highlights an important benefit of MemC3's approach of sharing all data between all threads.

5.2 SILT: On-Flash Key-Value Store

SILT (Small Index Large Table) [59] is a memory-efficient, high-performance key-value store system based on flash storage that scales to serve billions of key-value items on a single node. It requires only 0.7 bytes of DRAM per entry and retrieves key-value pairs using on average 1.01 flash reads each. SILT applies a set of new algorithmic and systems techniques to balance the requirement of memory, storage, and computation. SILT is work in conjunction with Hyeontaek Lim who designed and implemented the multi-store architecture and *entropy-coded trie* (ECT) [60] which requires only 0.4 bytes per key and is the primary source of the overall high space efficiency; this thesis constributed to SILT in the design and integration of partial-key cuckoo hash table and cuckoo filters to enable high-performance insertion and update at low memory cost.

5.2.1 SILT's Multi-Store Design

Goals and Rationale The design of SILT follows from five main goals:

1. **Low read amplification**: Issue at most $1 + \epsilon$ flash reads for a single `GET`, where ϵ is configurable and small (e.g., 0.01).
   Rationale: Random reads remain the read throughput bottleneck when using flash memory. Read amplification therefore directly reduces throughput.

2. **Controllable write amplification and favoring sequential writes**: It should be possible to adjust how many times a key-value entry is rewritten to flash over its lifetime. The system should issue flash-friendly, large writes.
   Rationale: Flash memory can undergo only a limited number of erase cycles before it fails. Random writes smaller than the SSD log-structured page size (typically 4 KiB[3]) cause extra flash traffic.

   Optimizations for memory efficiency and garbage collection often require data layout changes on flash. The system designer should be able to select an appropriate balance of flash lifetime, performance, and memory overhead.

[3] For clarity, binary prefixes (powers of 2) will include "i", while SI prefixes (powers of 10) will appear without any "i".

3. **Memory-efficient indexing**: SILT should use as little memory as possible (e.g., less than one byte per key stored).
   Rationale: DRAM is both more costly and power-hungry per gigabyte than Flash, and its capacity is growing more slowly.

4. **Computation-efficient indexing**: SILT's indexes should be fast enough to let the system saturate the flash I/O.
   Rationale: System balance and overall performance.

5. **Effective use of flash space**: Some data layout options use the flash space more sparsely to improve lookup or insertion speed, but the total space overhead of any such choice should remain small – less than 20% or so.
   Rationale: SSDs remain relatively expensive.

Conventional Single-Store Approach A common approach to building high-performance key-value stores on flash uses three components:

1. *an in-memory filter* to efficiently test whether a given key is stored in this store before accessing flash;

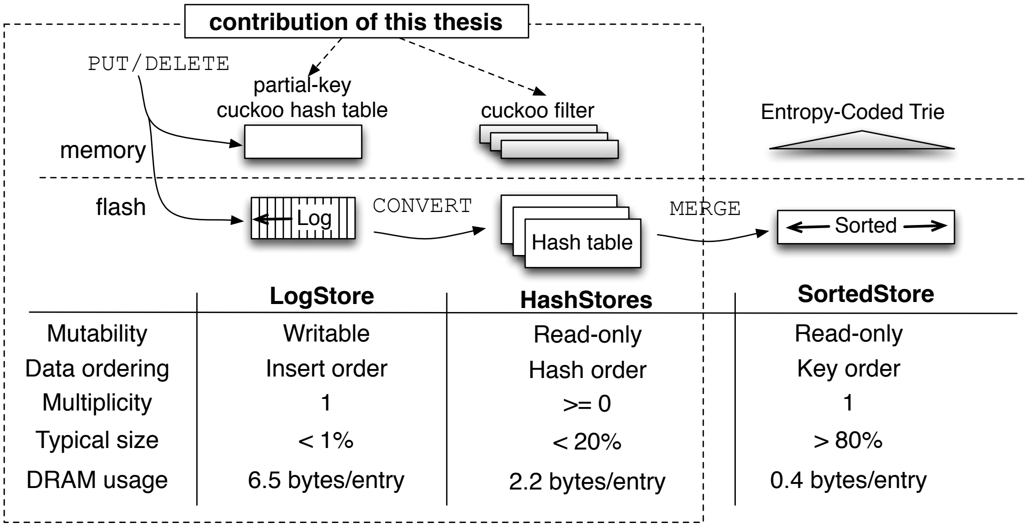
2. *an in-memory index* to locate the data on flash for a given key; and

3. *an on-flash data layout* to store all key-value pairs persistently.

Unfortunately, to our knowledge, no existing index data structure and on-flash layout achieve all of our goals simultaneously. For example, HashCache-Set [13] organizes on-flash keys as a hash table, eliminating the in-memory index, but incurring random writes that impair insertion speed. To avoid expensive random writes, systems such as FAWN-DS [9], FlashStore [34], and SkimpyStash [35] append new values sequentially to a log. These systems then require either an in-memory hash table to map a key to its offset in the log (often requiring 4 bytes of DRAM or more per entry) [9, 35]; or keep part of the index on flash using multiple random reads for each lookup [35].

Multi-Store Approach BigTable [24], Anvil [61], and BufferHash [8] chain multiple stores, each with different properties such as high write performance or inexpensive indexing.

Multi-store systems impose two challenges. First, they require effective designs and implementations of the individual stores: they must be efficient, compose well,



	LogStore	HashStores	SortedStore
Mutability	Writable	Read-only	Read-only
Data ordering	Insert order	Hash order	Key order
Multiplicity	1	>= 0	1
Typical size	< 1%	< 20%	> 80%
DRAM usage	6.5 bytes/entry	2.2 bytes/entry	0.4 bytes/entry

Figure 5.5: Overview of SILT and the contribution of this thesis.

and it must be efficient to transform data between the store types. Second, it must be efficient to query multiple stores when performing lookups. The design must keep read amplification low by not issuing flash reads to each store. A common solution uses a compact in-memory filter to test whether a given key can be found in a particular store, but this filter can be memory-intensive—e.g., BufferHash uses 4–6 bytes for each entry.

SILT's multi-store design uses a series of *basic key-value stores*, each optimized for a different purpose.

1. Keys are inserted into a write-optimized store, and over their lifetime flow into increasingly more memory-efficient stores.

2. Most key-value pairs are stored in the most memory-efficient basic store. Although data outside this store uses less memory-efficient indexes (e.g., to optimize writing performance), the average index cost per key remains low.

3. SILT is tuned for high worst-case performance—a lookup found in the last and largest store. As a result, SILT can avoid using an in-memory filter on this last store, allowing all lookups (successful or not) to take $1 + \epsilon$ flash reads.

SILT's architecture and basic stores (the LogStore, HashStore, and SortedStore) are depicted in Figure 5.5. Each store places different emphasis on memory-efficiency and write-friendliness; chaining all three stores create a system that provides high write speed, high read throughput, and uses little memory. This thesis describes the design and implementation of LogStore in Section 5.2.2 and HashStore in Section 5.2.2, which are built on partial-key cuckoo hashing and cuckoo filters introduced in previous Chapters. The theory of SortedStore and its applications in addition to SILT are presented in [60].

Key-Value Operations Each `PUT` operation inserts a `(key,value)` pair into the LogStore, even if the key already exists. `DELETE` operations likewise append a "delete" entry into the LogStore. The space occupied by deleted or stale data is reclaimed when SILT merges HashStores into the SortedStore. These lazy deletes trade flash space for sequential write performance.

To handle `GET`, SILT searches for the key in the LogStore, HashStores, and SortedStore in sequence, returning the value found in the youngest store. If the "deleted" entry is found, SILT will stop searching and return "not found."

5.2.2 LogStore and HashStore

LogStore

LogStore is a write-friendly key-value store that handles individual `PUT`s and `DELETE`s. This store consists of an on-flash data log and in-memory hash table index same as the FAWN-DS [9], but the in-memory index is implemented by a partial-key cuckoo hash table to improve its memory space utilization from 50% in FAWN-DS to 95%. To achieve high performance, all writes are appended to the end of a log file on flash. Because these items are ordered by their insertion time, the LogStore uses the in-memory hash table to map each key to its offset in the log file. SILT uses a memory-efficient, high-performance hash table based upon partial-key cuckoo hash table as described in Chapter 3. This index efficiently maps keys to their location in the flash log as shown in Figure 5.6, with 95% occupancy and very low computation and memory overhead. This is a substantial improvement over earlier systems such as FAWN-DS and BufferHash that achieved only 50% hash table occupancy. Compared to the other two read-only store types, however, this index is still relatively memory-intensive, because it must store one 4-byte pointer for every key. SILT therefore

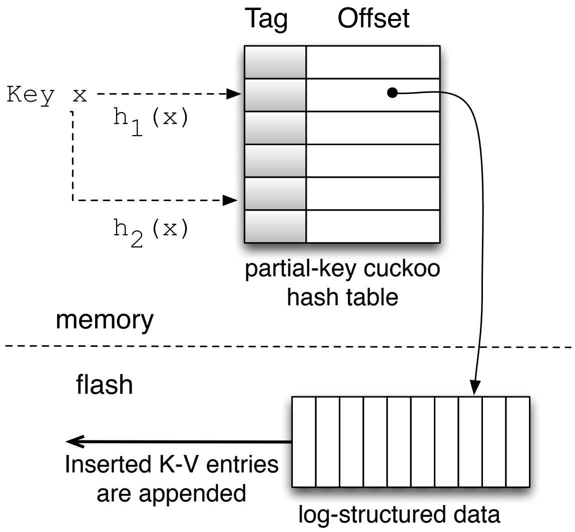

Tag
Offset
Key x
h1(x)
h2(x)
partial-key cuckoo hash table
memory
flash
Inserted K-V entries are appended
log-structured data

Figure 5.6: Design of LogStore: an in-memory cuckoo hash table (index and filter) and an on-flash data log.

uses only one instance of the LogStore (except during conversion to HashStore as described below), with fixed capacity to bound its memory consumption.

To keep low memory consumption, LogStore only stores the entire keys on the flash. Because retrieving the keys from flash is too expensive, we use partial-key cuckoo hashing instead of standard cuckoo hashing to build the key-value index. Thus a short "tag" (or fingerprint) instead of the actual key is stored in the hash table as shown in Figure 5.6. This gives LogStore two advantages in performance:

- It speeds up lookup performance because a lookup may check multiple buckets whereas it only proceeds to flash when the given key matches the tag, which can prevent most unnecessary flash reads for non-existing keys. If the tag matches, the full key and its value are retrieved from the log on flash to verify if the key it read was indeed the correct key.

- In addition, these tags enable the hash table to perform cuckoo insert without retrieving the full keys stored on flash. Standard cuckoo hashing moves existing keys to their alternative buckets and may in turn displace more keys; and each displacement required by cuckoo hashing would result in a flash reads. Using partial-key cuckoo hashing, the alternative buckets is derived using the current

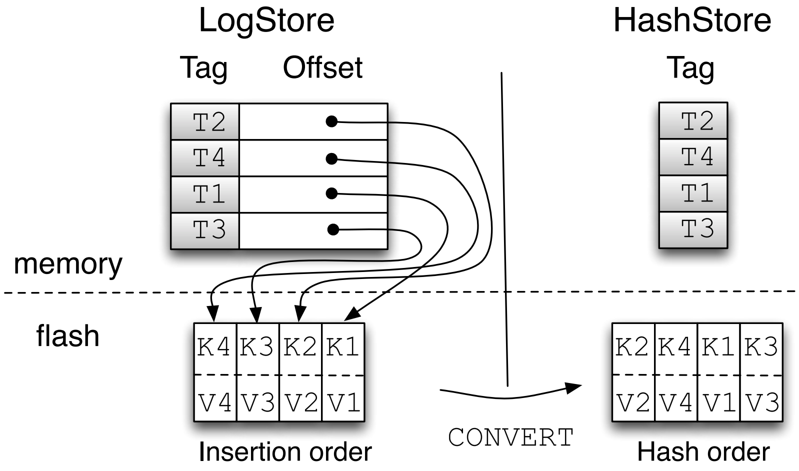


Figure 5.7: Convert a LogStore to a HashStore. Four keys K1, K2, K3, and K4 are inserted to the LogStore whose tags are T1, T2, T3 and T4, so the layout of the log file is the insert order; the in-memory index keeps the offset of each key on flash. In HashStore, the on-flash data forms a hash table where keys are in the same order as the in-memory filter.

bucket index and the tag, without requiring to retrieve the original key for rehashing.

By default, SILT uses a 15-bit key fragment as the tag. Each hash table entry is 6 bytes, consisting of a 15-bit tag, a single valid bit, and a 4-byte offset pointer. The probability of a false positive retrieval is 0.024%, i.e., on average 1 in 4,096 flash retrievals is unnecessary.

HashStore

Once a LogStore fills up (e.g., the insertion algorithm terminates without finding any vacant slot after a maximum number of displacements in the hash table), SILT freezes the LogStore and converts it into a more memory-efficient data structure. Directly sorting the relatively small LogStore and merging it into the much larger SortedStore requires rewriting large amounts of data, resulting in high write amplification. On the other hand, keeping a large number of LogStores around before merging could amortize the cost of rewriting, but unnecessarily incurs high memory overhead from the LogStore's index. To bridge this gap, SILT first converts the LogStore to an immutable HashStore with higher memory efficiency; once SILT accumulates a configurable number of HashStores, it performs a bulk merge to incorporate them into the SortedStore. During the LogStore to HashStore conversion, the old LogStore continues to serve lookups, and a new LogStore receives inserts.

HashStore saves memory over LogStore by eliminating the index and reordering the on-flash `(key,value)` pairs from insertion order to hash order (see Figure 5.7). HashStore is thus an on-flash cuckoo hash table, and has the same occupancy (93%) as the in-memory version found in LogStore. HashStores also have one in-memory component, a filter to probabilistically test whether a key is present in the store without performing a flash lookup.

Although prior approaches [8] used Bloom filters [19] for the probabilistic membership test, SILT uses a cuckoo filter based on partial-key cuckoo hashing. Cuckoo filters are more memory-efficient than Bloom filters at low false positive rates. Given a 15-bit tag in a 4-way set associative cuckoo hash table, the false positive rate is $f = 2^{-12} = 0.024\%$. With 95% table occupancy, the effective number of bits per key using a hash filter is $15/0.95 = 15.78$. In contrast, a standard Bloom filter that sets its number of hash functions to optimize space consumption requires at least $1.44 \log_2(1/f) = 17.28$ bits of memory to achieve the same false positive rate.

For HashStores, cuckoo filters are also efficient to create: SILT simply copies the tags from the LogStore's hash table, in order, discarding the offset pointers; on the contrary, Bloom filters would have been built from scratch, hashing every item in the LogStore again.

Alternative Designs Our current implementation of HashStore reads the log data of LogStore from flash, flushes the hash table to flash again. This conversion costs flash I/O bandwidth to save memory by eliminating the offset pointers. One alternative approach is to keep the log file from LogStore without change, keep the tags in memory, but flush the pointers to flash. This approach reduces the flash I/O traffic, however requires one more flash read to lookup the offset before retrieving the data in the log, and thus increases the `GET` latency.

5.2.3 Evaluation

This section evaluates the performance of LogStore and HashStore.

Evaluation System The testbed runs Linux using equipped with:

Type	**Speed (K keys/s)**
LogStore (by PUT)	204.6
HashStore (by CONVERT)	67.96

Table 5.3: Construction performance for basic stores. The construction method is shown in the parentheses.

Type	**HashStore**	**LogStore**
	(K ops/s)	**(K ops/s)**
GET (hit)	44.93	46.79
GET (miss)	7264	7086

Table 5.4: Query performance for basic stores that include in-memory and on-flash data structures.

CPU	Intel Core i7 860 @ 2.80 GHz (4 cores)
DRAM	DDR SDRAM / 8 GiB
SSD-L	Crucial RealSSD C300 / 256 GB
SSD-S	Intel X25-E / 32 GB

The 256 GB SSD-L stores the key-value data, and the SSD-S is used as scratch space for sorting HashStores using Nsort [75]. The drives connect using SATA and are formatted with the ext4 filesystem using the discard mount option (TRIM support) to enable the flash device to free blocks from deleted files. The baseline performance of the data SSD is:

Random Reads (1024 B)	48 K reads/sec
Sequential Reads	256 MB/sec
Sequential Writes	233 MB/sec

Individual Store Microbenchmark Here we measure the performance of each SILT store type in its entirety (in-memory indexing plus on-flash I/O). The first experiment builds multiple instances of each basic store type with 100 M key-value pairs (20-byte key, 1000-byte value). The second experiment queries each store for 10 M random keys.

Table 5.3 shows the construction performance for these two stores; the construction method is shown in parentheses. LogStore construction, built through entry-by-entry insertion using PUT, can use 90% of sequential write bandwidth of the flash drive. Thus, SILT is well-suited to handle bursty inserts. The conversion from LogStores to HashStores is about three times slower than LogStore construction because it involves bulk data reads and writes from/to the same flash drive. SortedStore construction is slowest, as it involves an external sort for the entire group of 31 HashStores to make one SortedStore (assuming no previous SortedStore). If constructing the SortedStore involved merging the new data with an existing SortedStore, the performance would be worse. The large time required to create a SortedStore was one of the motivations for introducing HashStores rather than keeping un-merged data in LogStores.

Table 5.4 shows that the minimum `GET` performance is 44.93 K ops/s at HashStores. Note that LogStores and HashStores are particularly fast at `GET` for nonexistent keys (more than 7 M ops/s).

5.3 Related Work

This section presents work most related to improve individual key-value storage nodes in terms of throughput and space efficiency.

Flash-based Key-Value Stores BufferHash [8], FAWN-DS [10], and SkimpyStash [35] are optimized for I/O to external storage such as SSDs (e.g., by batching, or log-structuring small writes). The more relevant work to SILT is BufferHash that keeps keys in multiple equal-sized hash tables—one in memory and the others on flash. The on-flash tables are guarded by in-memory Bloom filters to reduce unnecessary flash reads. In contrast, SILT data stores have different sizes and types. The largest store (SortedStore), for example, does not have a filter and is accessed at most once per lookup, which saves memory while keeping the read amplification low. In addition, writes in SILT are appended to a log stored on flash for crash recovery, whereas inserted keys in BufferHash do not persist until flushed to flash in batch.

Several key-value storage libraries rely on caching to compensate for their high read amplifications [17] [46], making query performance depend greatly on whether the working set fits in the in-memory cache. In contrast, SILT provides uniform and predictably high performance regardless of the working set size and query patterns.

Memory-based Key-Value Stores [16, 50, 66] boost performance on multi-core CPUs or GP-GPUs by sharding data to dedicated cores to avoid synchronization.

MemC3 instead targets read-mostly workloads and deliberately avoids sharding to ensure high performance even for “hot” keys. Similar to MemC3, Masstree [63] also applied extensive optimizations for cache locality and optimistic concurrency control, but used very different techniques because it was a variation of B+-tree to support range queries. RAMCloud [78] focused on fast data reconstruction from on-disk replicas. In contrast, as a cache, MemC3 specifically takes advantage of the transience of the data it stores to improve space efficiency.

Modular Design of Storage Systems BigTable [24] and Anvil [61] both provide a modular architecture for chaining specialized stores to benefit from combining different optimizations. SILT borrows its design philosophy from these systems; we believe and hope that the techniques we developed for SILT could also be used within these frameworks.

Chapter 6

Conclusion

This dissertation proposed, analyzed and evaluated several more-efficient key-value storage systems, whose design is grounded in recent theory and informed by the underlying hardware and expected workloads. Our systems achieve higher efficiency by (1) improving each individual key-value node in better performance and less resource consumption; and (2) making near-optimal capacity utilization across many nodes for unfriendly or even adversarial workloads. This dissertation made the following contributions:

- Through analysis, simulation, and experiments on an 85-node cluster, we have demonstrated that a small, fast front-end cache can ensure effective load-balancing, regardless of the query distribution. We have proven a lower bound on the cache size that depends only on the number of back-end nodes in the system, not the number of items stored.

- We presented two new data structures that help build more efficient key-value stores. The first is *cuckoo filters* for approximate set-membership queries. Cuckoo filters improve upon Bloom filters in three ways: (1) support for deleting items dynamically; (2) better lookup performance; and (3) better space efficiency for applications requiring low false positive rates ($\epsilon < 3\%$). The second is optimistic cuckoo hashing, a new hashing scheme that achieves over 90% space occupancy and allows concurrent read access without locking.

- Combining new algorithmic and systems techniques, we implemented MemC3, an in-memory key-value cache that reduced the space overhead by more than 20 bytes per entry but performed 3× faster than the original Memcached while storing 30% more objects for small key-value pairs; we also designed and implemented SILT, a high-speed on-flash key-value store that cost 0.7 bytes of memory for entry it stored, and made only 1.01 flash reads to service a lookup, doing so in under 400 microseconds.

In summary, this dissertation demonstrated that careful attention to algorithm and data structure design can significantly improve throughput and memory efficiency for key-value systems. In addition, I believe the techniques and insights developed by this work will apply generally to other distributed and networking systems.

Bibliography

[1] CityHash. https://code.google.com/p/cityhash/. [Cited on page 70.]

[2] Hadoop. http://hadoop.apache.org/, 2011. [Cited on page 1.]

[3] HBase. http://hbase.apache.org/, 2011. [Cited on pages 7 and 49.]

[4] Intel Threading Building Blocks. http://threadingbuildingblocks.org/, 2011. [Cited on page 11.]

[5] Apache Thrift. https://thrift.apache.org/, 2011. [Cited on page 11.]

[6] Aerospike. http://www.aerospike.com/. [Cited on page 7.]

[7] Hrishikesh Amur, Wolfgang Richter, David G. Andersen, Michael Kaminsky, Karsten Schwan, Athula Balanachandran, and Erik Zawadzki. Memory-efficient groupby-aggregate using compressed buffer trees. In *Proc. 4rd ACM Symposium on Cloud Computing (SOCC)*, Santa Clara, CA, October 2013. [Cited on page 2.]

[8] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, April 2010. [Cited on pages 2, 93, 98 and 100.]

[9] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on pages 2, 5, 7, 10, 11, 28, 93 and 95.]

[10] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. *Communications of the ACM*, 54(7):101–109, July 2011. [Cited on pages 2, 3 and 100.]

[11] I. Ari, B. Hong, E.L. Miller, S.A. Brandt, and D.D.E. Long. Managing flash crowds on the Internet. In *Proceedings of 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, pages 246–249, October 2003. [Cited on page 5.]

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012. [Cited on page 88.]

[13] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: cache storage for the next billion. In *Proc. 6th USENIX NSDI*, Boston, MA, April 2009. [Cited on pages 2 and 93.]

[14] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010. [Cited on pages 1 and 7.]

[15] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012. [Cited on pages 53, 54, 70 and 79.]

[16] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *In Proceedings of the Second International Green Computing Conference*, Orlando, FL, August 2011. [Cited on pages 84 and 100.]

[17] BerkeleyDB Reference Guide. Memory-only or Flash configurations. `http://www.oracle.com/technology/documentation/berkeley-db/db/ref/program/ram.html`. [Cited on page 100.]

[18] A. Bestavros. WWW traffic reduction and load balancing through server-based caching. *Concurrency, IEEE*, 5(1):56–67, 1997. [Cited on page 28.]

[19] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. [Cited on pages 4, 49, 52, 69 and 98.]

[20] bobhash. Bob Jenkins Hash. `http://www.burtleburtle.net/bob/c/lookup3.c/`, 2006. [Cited on pages 4 and 70.]

[21] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. Bloom filters via d-left hashing and dynamic bit reassignment. In *Proceedings of the Allerton Conference on Communication, Control and Computing*, 2006. [Cited on pages 53, 65, 70 and 79.]

[22] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *14th Annual European Symposium on Algorithms, LNCS 4168*, pages 684–695, 2006. [Cited on pages 53 and 79.]

[23] Andrei Broder, Michael Mitzenmacher, and Andrei Broder. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, volume 1, pages 636–646, 2002. [Cited on pages 49, 51 and 65.]

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Seattle, WA, November 2006. [Cited on pages 1, 5, 7, 49, 93 and 101.]

[25] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, Ayellet Tal, and Oh Boy. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of SODA*, pages 30–39, 2004. [Cited on page 69.]

[26] cityhash. CityHash. `http://code.google.com/p/cityhash/`, 2013. [Cited on page 4.]

[27] John G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Transactions on Computer*, C-33(9), September 1984. [Cited on page 80.]

[28] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010. [Cited on page 87.]

[29] F.J. Corbato and MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC. *A Paging Experiment with the Multics System.* Defense Technical Information Center, 1968. URL `http://books.google.com/books?id=5wDQNwAACAAJ`. [Cited on page 85.]

[30] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001. [Cited on pages 1, 19 and 27.]

[31] Jeffrey Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, New York, NY, USA, 2009. ACM. [Cited on page 28.]

[32] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013. [Cited on page 2.]

[33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004. [Cited on pages 1 and 5.]

[34] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010. [Cited on pages 2 and 93.]

[35] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM space skimpy key-value store on flash. In *Proc. ACM SIGMOD*, Athens, Greece, June 2011. [Cited on pages 2, 93 and 100.]

[36] Guiseppe DeCandia, Deinz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007. [Cited on pages 1, 2, 5, 7, 10, 11 and 27.]

[37] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1):47–68, 2007. [Cited on pages 31, 34 and 57.]

[38] U. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Proc. Seventh Workshop on Distributed Data and Structures (WDAS'06)*, CA, USA, January 2006. [Cited on pages 29, 31 and 66.]

[39] Facebook Engineering Notes. Scaling memcached at Facebook. http://www.facebook.com/note.php?note_id=39391378919. [Cited on page 82.]

[40] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proc. 2nd ACM Symposium on Cloud Computing (SOCC)*, Cascais, Portugal, October 2011. [Cited on pages viii and ix.]

[41] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013. [Cited on pages 4 and 81.]

[42] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. ACM SIGCOMM*, Vancouver, BC, Canada, September 1998. [Cited on pages 52 and 79.]

[43] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010. [Cited on page 1.]

[44] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1222–1236. SIAM, 2011. [Cited on pages 31, 34 and 57.]

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, October 2003. [Cited on page 1.]

[46] Google. LevelDB. `https://code.google.com/p/leveldb/`. [Cited on pages 49 and 100.]

[47] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th USENIX OSDI*, San Diego, CA, November 2000. [Cited on page 10.]

[48] Neil Gunther, Shanti Subramanyam, and Stefan Parvu. Hidden scalability gotchas in memcached and friends. In *VELOCITY Web Performance and Operations Conference*, Santa Clara, CA, USA, June 2010. [Cited on page 84.]

[49] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914. [Cited on pages 35, 36 and 38.]

[50] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012. [Cited on pages 84 and 100.]

[51] Nan Hua, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu. Rank-Indexed Hashing: A Compact Construction of Bloom Filters and Variants. In *Proc. of IEEE Int'l Conf. on Network Protocols (ICNP) 2008*, Orlando, Florida, USA, October 2008. [Cited on page 79.]

[52] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. 2nd ACM European Conference on Computer Systems (EuroSys)*, Lisboa, Portugal, March 2007. [Cited on page 1.]

[53] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. [Cited on page 11.]

[54] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM. [Cited on pages 1, 5, 19 and 27.]

[55] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms*, 33(2):187–218, 2008. [Cited on page 69.]

[56] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981. ISSN 0362-5915. [Cited on page 38.]

[57] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978. [Cited on page 27.]

[58] libMemcached. libmemcached. `http://libmemcached.org/`, 2009. [Cited on page 91.]

[59] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011. [Cited on pages viii, ix, 2, 4, 81 and 92.]

[60] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Practical batch-updatable external hashing with sorting. In *Proc. Meeting on Algorithm Engineering and Experiments (ALENEX)*, January 2013. [Cited on pages 92 and 95.]

[61] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with Anvil. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on pages 93 and 101.]

[62] Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. In *Information Processing Letters*, pages 50–92, 1994. [Cited on page 80.]

[63] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, April 2012. [Cited on page 101.]

[64] E. P Markatos. On caching search engine query results. *Computer Communications*, 24(2):137 – 143, 2001. ISSN 0140-3664. [Cited on page 28.]

[65] Memcached. A distributed memory object caching system. `http://memcached.org/`, 2011. [Cited on pages 7, 10, 28 and 82.]

[66] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2012. [Cited on pages 84 and 100.]

[67] M. Mitzenmacher and B. Vocking. The asymptotics of selecting the shortest of two, improved. In *Proc. the Annual Allerton Conference on Communication Control and Computing*, volume 37, pages 326–327, 1999. [Cited on page 53.]

[68] Michael Mitzenmacher. Some open questions related to cuckoo hashing, 2009. [Cited on page 4.]

[69] Michael Mitzenmacher, AndrÃľa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000. [Cited on pages 16 and 72.]

[70] MongoDB. `http://mongodb.com`. [Cited on page 82.]

[71] murmurhash. MurmurHash. `https://sites.google.com/site/murmurhash/`, 2011. [Cited on page 70.]

[72] Suman Nath and Aman Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of ACM/IEEE International Conference on Information Processing in Sensor Networks*, Cambridge, MA, April 2007. [Cited on page 2.]

[73] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013. [Cited on page 2.]

[74] NIST. Secure Hash Standard (SHS). In *FIPS Publication 180-1*, 1995. [Cited on page 4.]

[75] C. Nyberg and C. Koester. Ordinal Technology - Nsort home page. `http://www.ordinal.com`, 2012. [Cited on page 99.]

[76] O. O'Malley and A.C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. `http://sortbenchmark.org/Yahoo2009.pdf`, April 2009. [Cited on page 1.]

[77] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996. [Cited on page 49.]

[78] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011. [Cited on page 101.]

[79] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar,

Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. In *Operating Systems Review*, volume 43, pages 92–105, January 2010. [Cited on pages 11 and 28.]

[80] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *In Proc. of ASM-SIAM Symposium on Discrete Algorithms, SODA 2005*, 2005. [Cited on pages 52 and 80.]

[81] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004. [Cited on pages 3, 29, 30 and 31.]

[82] Niels Provos. libevent. `http://monkey.org/~provos/libevent/`. [Cited on page 84.]

[83] Felix Putze, Peter Sanders, and Singler Johannes. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms*, pages 108–121. Springer Berlin / Heidelberg, 2007. [Cited on pages 53, 70 and 80.]

[84] Martin Raab and Angelika Steger. “Balls into bins” — a simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM ’98, pages 159–170, London, UK, 1998. Springer-Verlag. [Cited on page 16.]

[85] Redis. `http://redis.io`. [Cited on pages 7 and 82.]

[86] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In K. SelÃğuk Candan, Yi Chen 0001, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *SIGMOD Conference*, pages 217–228. ACM, 2012. [Cited on page 49.]

[87] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001. [Cited on pages 1, 5, 11, 19 and 27.]

[88] VoltDB. VoltDB, the NewSQL database for high velocity applications. `http://voltdb.com/`. [Cited on page 82.]

[89] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. MicroHash: An efficient index structure for flash-based

sensor devices. In *Proc. 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, December 2005. [Cited on page 2.]