Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services

Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky[*]
Carnegie Mellon University, [*]Intel Labs
{binfan,hl,dga}@cs.cmu.edu, michael.e.kaminsky@intel.com

ABSTRACT

Load balancing requests across a cluster of back-end servers is critical for avoiding performance bottlenecks and meeting service-level objectives (SLOs) in large-scale cloud computing services. This paper shows how a small, fast popularity-based front-end cache can ensure load balancing for an important class of such services; furthermore, we prove an $O(n \log n)$ lower-bound on the necessary cache size and show that this size depends only on the total number of back-end nodes n, not the number of items stored in the system. We validate our analysis through simulation and empirical results running a key-value storage system on an 85-node cluster.

CATEGORIES AND SUBJECT DESCRIPTORS

D.4.2 [**Operating Systems**]: Storage Management; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

GENERAL TERMS

Design, Measurement, Performance

KEYWORDS

Caching, load balancing, clusters, performance

1. INTRODUCTION

As data intensive computing grows in both popularity and in scale [15, 14, 11], load balancing—across thousands of nodes and beyond—grows simultaneously more important and more challenging. Today, numerous companies operate storage and processing clusters at this scale, with familiar examples including Google's BigTable and GFS cells (1000 to 7000 nodes in one cluster [17]), Facebook's photo storage (20 petabytes of data [8]), Microsoft's data mining cluster (1800 nodes [21]), and Yahoo's Hammer cluster (3800 nodes [27]). As a result, system designers must be ever more careful that as the clusters grow, their performance does not become bottlenecked due to unevenly partitioned load among cluster nodes. Many of these services must further cope with potentially unpredictable shifts in the query workload (i.e., "flash crowds" [7]) or adversarial access patterns, either accidentally or as a denial-of-service attack.

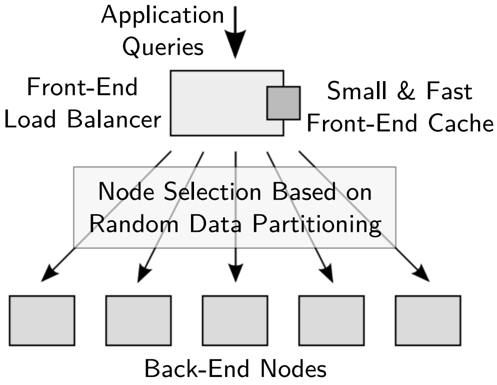

Application Queries
Front-End Load Balancer
Small & Fast Front-End Cache
Node Selection Based on Random Data Partitioning
Back-End Nodes

Figure 1: Small, fast cache at the front-end load balancer.

This paper shows that system designers can ensure load balancing for an important class of services 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") in cluster architectures such as that shown in Figure 1. This cache[1] is small enough to fit in the L3 cache of a fast CPU, enabling an efficient, high-speed implementation compatible with front-end load balancers and packet processors. Our work exploits 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.

We begin by proving that the cache need only store $O(n \log n)$ entries to provide good load balance, where n is the total number of back-end nodes. As a concrete example, a key-value storage system with 100 nodes using 1 KiB entries can be serviced using 4 megabytes of fast CPU cache memory, regardless of the query distribution that it must handle. This result enables constructing clusters that use large numbers of slower, but more energy- and cost-efficient nodes to provide massive storage and high overall throughput, coupled with a small number of fast machines operating in their ideal performance range: serving a high query rate with all of their code and data in the CPU cache.

[1] Unless mentioned specifically, we use the term "cache" to refer to an *application-level* cache, not a CPU cache.

SOCC'11, October 27–28, 2011, Cascais, Portugal.
Copyright 2011 ACM 978-1-4503-0976-9/11/10...$10.00.

Our result applies to a class of services that are popular building blocks for several distributed systems. We target services 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 file systems** such as the Google File System (GFS) [18] and the Hadoop Distributed File System (HDFS) [1], where each data block is located and served at one or multiple semi-random servers;
- **Distributed object caches** such as memcached [25];
- **Distributed key-value storage systems** such as Dynamo [15], Haystack [8], and FAWN-KV [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 [11] and HBase [2] 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.

Our overall goal is to allow cloud service providers to meet service-level objectives (SLOs) for handling a particular rate of queries regardless of the query distribution, without the need for drastic over-provisioning. Our analytical results in Section 4 show that a small front-end cache can do exactly this. Furthermore, we provide guidance for provisioning such a cache by showing tight analytical bounds on the necessary cache size. Importantly, we find that the cache size depends only on the total number of back-end storage nodes, not the number of items stored in the system.

Finally, we validate these results empirically using an 85-node testbed cluster using the FAWN-KV key-value storage system [6], presented in Section 5. Because of the tight bounds on the cache size, we are able to implement a front-end cache that fits in the 2×12MB of L3 cache available on two contemporary server CPUs. We show that even a simple userland implementation of the front-end cache can handle more than 800,000 queries per second running on two low-power 2.27GHz Xeon processors.

2. BACKGROUND & CONTEXT

Cluster systems scale using a combination of partitioning (spreading data or responsibility across a larger number of nodes, where each node handles a different subset of the requirements) and replication. For some services, such as partitioned search, the response time of the cluster is equal to the response time of the slowest node to respond to a query. For others, such as partitioned key-value stores, uneven load degrades service for the fraction of the data handled by the overloaded nodes. In both cases, good load balance is necessary to ensure that the cluster can meet its throughput and latency goals.

Systems must balance both 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 handle—and the *dynamic* load of handling queries as they arrive. Data should be spread uniformly among nodes, and no node should handle too many more queries than another node (proportional to its relative ability/capacity).

Capacity is typically load-balanced by striping deterministically (e.g., RAID [29]) with carefully chosen boundaries for stripes, or by using a hashing-based approach. Consistent hashing schemes (e.g., that used in Chord [23, 32]) are popular due to their simplicity and ability to support incremental growth. Many systems that use consistent hashing use “virtual nodes” to improve the quality of their static load balance, where each physical server acts as several different nodes in the consistent hashing ring [12].

Although these schemes help balance the static space utilization, they do not balance the dynamic load: hotspots can still occur when some items are queried for more than others. Unfortunately, many real-world workloads have non-uniform query distributions.

Systems typically balance dynamic load in one of two ways. First, some dynamically move data from busy nodes to less busy nodes to help even the query load [31]. Others, such as Mitzenmacher’s well-known “power of two choices” load balancing, rely upon replication to be able to direct queries to the least-loaded of two or more replicas, substantially improving load balance in the process [26]. Unfortunately, both such schemes are limited to handling relatively small load imbalance (where no object is too hot to be served by one or a small, constant number of servers), and both introduce either consistency and migration challenges or high space overhead for full replication. This is not to say that these techniques are unnecessary—we discuss further in Section 6 how two-choices load balancing might be used synergistically with small-fast-cache load balancing.

Some large scale systems have applied combinations of these techniques, e.g., Amazon’s Dynamo [15] 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 the time (and provide less detail about load spikes or adversarial workloads). Load imbalance remains an important challenge for partitioned services [13].

3. SYSTEM MODEL

Before presenting our analytical results (Section 4), we first introduce our system model through an example—the FAWN-KV [6] key-value storage system—which we use for our experimental results in Section 5. FAWN-KV is a distributed high-performance key-value storage system. Like other key-value hash tables such as Dynamo [15], memcached [25], Citrusleaf, and cluster distributed hash tables [19], FAWN-KV provides a simple hashtable-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 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 [28]. 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 [15].

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 [6], but has less headroom for handling query bursts or popularity shifts [22]. 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 [32]). 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 & 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 system (L3) cache can satisfy these two requirements to act as a front-end for a cluster of Atom-based FAWN nodes with SSDs. Our software-based front-end cache uses a widely available hash table implementation [3] that supports concurrent access from multiple threads. The system uses Thrift [4] for marshaling, unmarshaling, and sending RPCs between the front-end and back-ends.

4. ANALYSIS

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.

Symbol	Meaning
n	# of back-end nodes
m	# of (key,value) items stored in the system
c	# of (key,value) items cached
R	sustainable query rate
L_i	query rate going to node i
r_i	max query rate supported by node i
p_i	fraction of queries for the ith (key, value)

Table 1: Notation used for the analysis.

4.1 Model and Assumptions

Table 1 summarizes the notation used in the analysis.

Model Consider a system such as that shown in Figure 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.

Assumptions This analysis makes four assumptions about the system. As the real systems may not necessarily obey these assumptions, we examine the effect of the factors that may affect load balancing in Section 5 and extend the discussion in Section 6.

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.

Promised 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.

Adversary 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 *can not easily target one specific node* as the hotspot, because it does not know which keys are assigned to which nodes.[2] In order to generate a skewed workload on the

[2] 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.

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

$$S = (p_1, p_2, \cdots, p_m), \tag{1}$$

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.,

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

4.2 Adversarial Access Pattern

We first examine the best strategy the adversary could adopt when the back-end servers have homogeneous capacity: $r_1 = \cdots = r_n = r$. We extend this result to heterogeneous nodes in Section 4.4.

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

$$S = (1, 0, \cdots, 0). \tag{2}$$

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.

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

$$S : \underbrace{p_1 \geq p_2 \geq \cdots \geq p_c}_{\text{cached keys}} \geq \underbrace{p_{c+1} \geq \cdots \geq p_m}_{\text{uncached keys}}. \tag{3}$$

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. $p_1 = p_2 = \cdots = p_c$):

$$S : \underbrace{p_1 = p_2 = \cdots = p_c}_{\text{cached keys}} = h \geq \underbrace{p_{c+1} \geq \cdots \geq p_m}_{\text{uncached keys}} \geq 0. \tag{4}$$

The following theorem states the best strategy for the adversary in terms of the uncached keys (see Figure 2):

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 $\delta = \min\{h - p_i, p_j\}$.*

The proof of this theorem is in Appendix A. 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

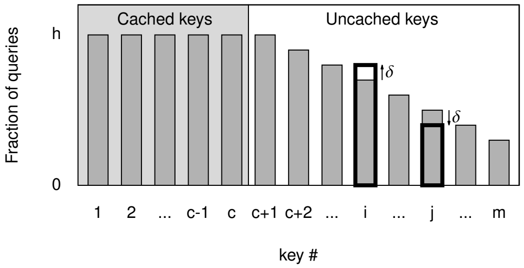


Figure 2: The construction of the best strategy for the adversary. The adversary can increase the expectation of the maximum load by moving query rate δ from a uncached key j to a more popular uncached key i (Theorem 1).

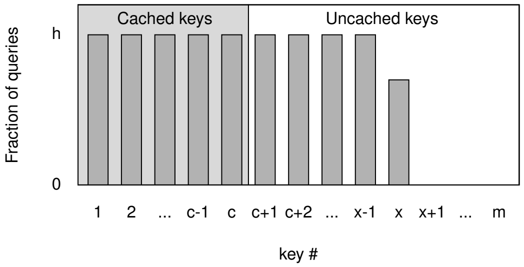


Figure 3: The constructed best strategy for the adversary to maximize load imbalance. The adversary queries x keys, where the rate for $x - 1$ is the same.

$$\begin{aligned} S: \quad & \underbrace{p_1 = \cdots = p_c}_{\text{cached keys}} = h = \underbrace{p_{c+1} = \cdots = p_{x-1} \geq p_x}_{\text{uncached keys}} > 0 \\ & p_{x+1} = \cdots = p_m = 0 \end{aligned} \tag{5}$$

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 3.

Note that Eq. (5) 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. In Appendix B, we derive an estimate of x^*, the optimal value of x, keeping in mind that the adversarially "optimal" number of keys to query against is the number that maximizes the load on one back-end node. We use this estimate in the next section to provide a lower bound on the throughput of a system with an appropriately-sized cache.

4.3 Throughput Bound

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 [30, 26]. Imagine we are throwing M identical balls into

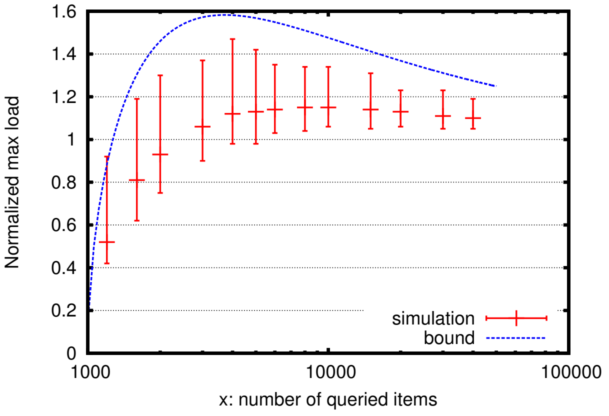


Figure 4: Simulation of the maximum number of keys on n = 100 nodes with cache size c = 1000, with 1000 runs.

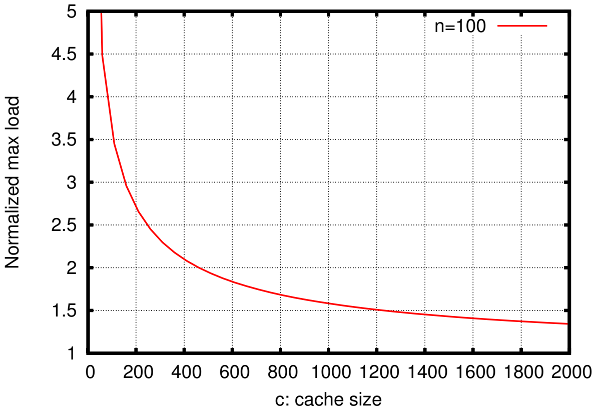


Figure 5: Normalized maximum load under optimal x as a function of the cache size c.

N bins and each bin is picked randomly. When $M \gg N \log N$, the number of balls in any bin is bounded by

$$\frac{M}{N} + \alpha\sqrt{2 \cdot \frac{M}{N} \cdot \log N}, \tag{6}$$

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 (6), the number of different keys served by any server is bounded by

$$\frac{x-c}{n} + \alpha\sqrt{\frac{2(x-c)}{n}\log n}. \tag{7}$$

Based on Eq. (7), 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. (7)), an upper bound of the expected load on each node is

$$\begin{aligned}\mathbb{E}[L_{\max}] &\leq \left(\frac{x-c}{n} + \alpha\sqrt{\frac{2(x-c)}{n}\log n}\right) \cdot \frac{R}{x-1} \\ &= \left(\frac{x-c}{x-1} + \alpha\sqrt{\frac{2(x-c)}{(x-1)^2} n \log n}\right)\frac{R}{n} \end{aligned} \tag{8}$$

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. We repeat this simulation for 1000 runs, and show the average, max and min of the maximum load in Figure 4. Each bar in the figure represents the maximum load obtained from 1000 runs with average, max, and min. The curve is calculated from Eq. (8) with $\alpha = 2$. The figure shows the bound has a small gap from the numerical results when $\alpha = 2$. Figure 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.

Based on Eq. (8), we can bound (see the derivation in the appendix) the worst case (the maximum point in Figure 4) for any x by

$$\mathbb{E}[L_{\max}] \leq \frac{1 + \sqrt{1 + 2\alpha^2 \frac{n \log n}{c-1}}}{2} \cdot \frac{R}{n}, \tag{9}$$

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

$$\frac{\mathbb{E}[L_{\max}]}{R/n} \leq \frac{1 + \sqrt{1 + 2\alpha^2 \frac{n \log n}{c-1}}}{2}. \tag{10}$$

Figure 5 illustrates the relationship between the max load (normalized) calculated in Eq. (10) 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 $O(n \log n)$ 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. (10) becomes constant in the system size:

$$\frac{1}{2}\left(1 + \sqrt{1 + \frac{2\alpha^2}{k}}\right) \tag{11}$$

Figure 6 shows the normalized load (Eq. (11) 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. (11) is independent of n, a system designer can choose a value for k that bounds the load (amount of overprovisioning required) at the back-ends. This normalized load/overprovisioning will be the same for any n, provided the front-end cache is scaled appropriately (to the value of c given above).

Promised Throughput Bound If the capacity r of each node is larger than the upper bound of $\mathbb{E}[L_{\max}]$ given in Eq. (9), then with high probability, the adversary can never saturate any node. The

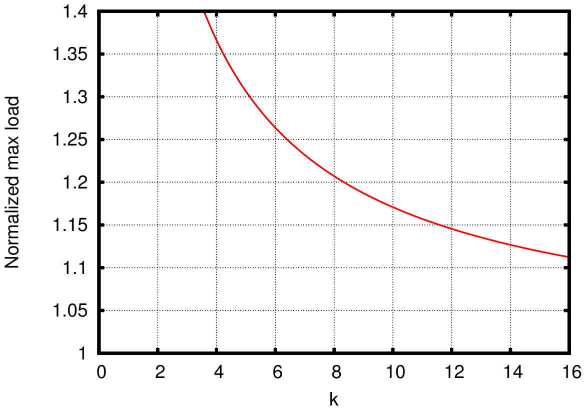


Figure 6: Normalized maximum load under optimal *x* as a function of *k*.

system can therefore guarantee that its throughput will always be equal or better than R queries per second.

$$R \geq \frac{2}{1 + \sqrt{1 + 2\alpha^2 \frac{n \log n}{c-1}}} \cdot n \cdot r, \qquad (12)$$

no matter what distribution of key queries the adversary uses.

4.4 Heterogeneous Nodes

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 [32, 23, 12]. 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. As we show in Appendix C, we can modify the preceding analysis to determine the highest load among all virtual nodes (instead of physical nodes), and then determine the load of a physical node based upon its virtual node count.

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.

Summary

- The worst case for the system (the best case for the adversary) is to send queries for x^* different items at an equal rate. x^* is a function of cache size c and cluster size n, but independent of the number of items stored in the cluster (see Appendix B).

	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: Specifications of front- and back-end nodes

- To achieve reasonably good load balance and avoid hotspots, the system needs a small, fast cache at the front-end of size $c = k \cdot n \log n + 1 = O(n \log n)$.
- k has a diminishing return with respect to improving load balancing. With heterogeneous nodes, k can be scaled approximately by the difference in capacity between the average and slowest nodes, and the analysis holds.

5. EVALUATION

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.

5.1 Experiment Overview

This section describes the cluster hardware and common experimental parameters.

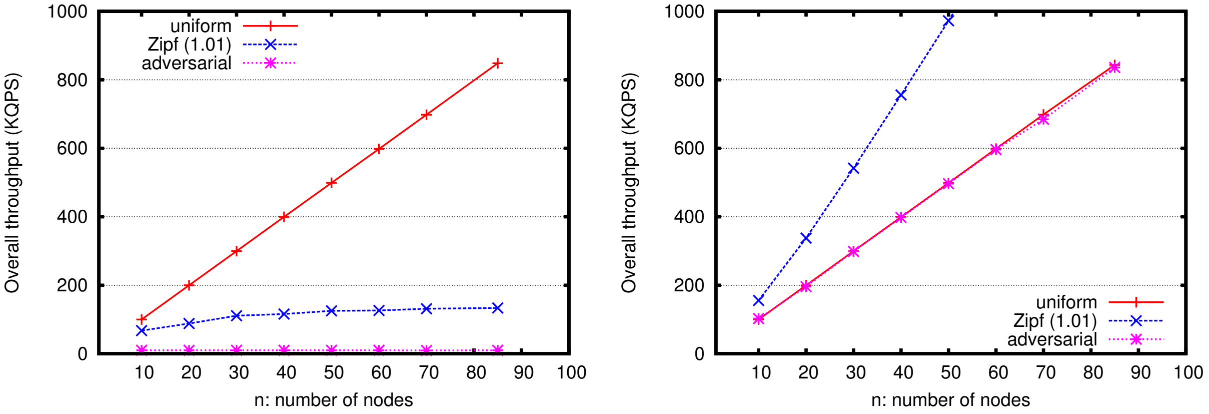
Experimental Testbed Our FAWN-KV cluster consists of one high-performance front-end node and 85 low-power back-end nodes. The front- and back-end node's specifications are shown in Table 2. All nodes are connected to a switch; the front-end node uses a 10 GbE link, while back-end nodes use 1 GbE links. 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.[3] 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.

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 approximately $10,000$ 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

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



(a) Without cache

(b) With cache

Figure 7: System throughput scalability as the number of back-end nodes increases from 10 to 85, under different access patterns including uniform, Zipf, and an adversarial workload.

serves 10,000 requests/second, to be able to scale our experiments to more nodes than we have SSDs for.

5.2 Caching Effect: Cluster Scaling Under Different Workloads

Figure 7 shows the scalability of the cluster throughput without and with caching as the number of back-end nodes in the cluster increases. We explore 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[4]; 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 7a 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 7b 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, 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.

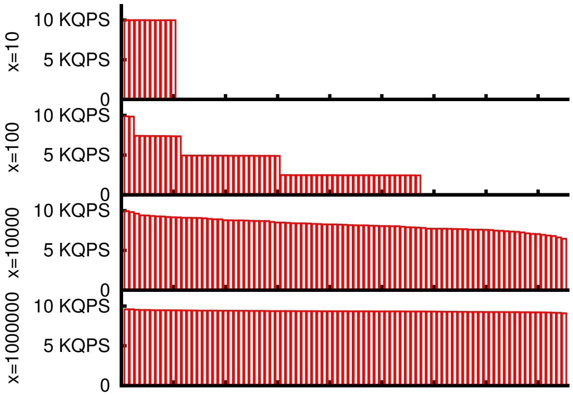


Figure 8: Throughput of each back-end node for different values of x when $c = 0$. Node IDs (x-axis) are sorted according to their throughput.

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

To further understand the interaction between load balancing and system throughput, Figure 8 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).

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

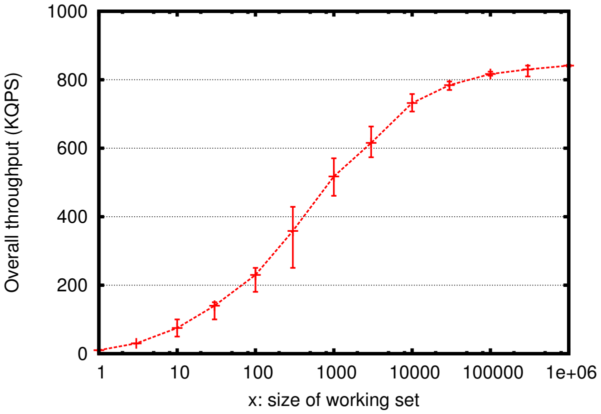


Figure 9: Throughput that the adversary can obtain by querying for different numbers of keys when the cluster has 85 back-end nodes and no cache is used.

- When $x = 10$, 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.
- 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.

5.4 Adversarial Throughput Under Different Query Patterns

Figure 9 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 10 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.

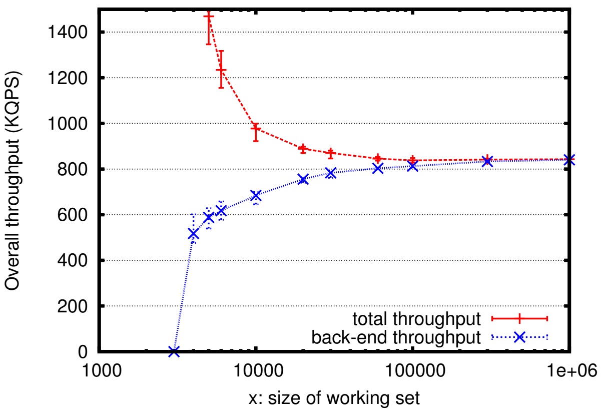


Figure 10: Throughput that the adversary can obtain by querying for different numbers of keys when the cluster has 85 back-end nodes and $c = 3000$. Results for $x < 3000$ are omitted because all queries would be served by the front-end cache.

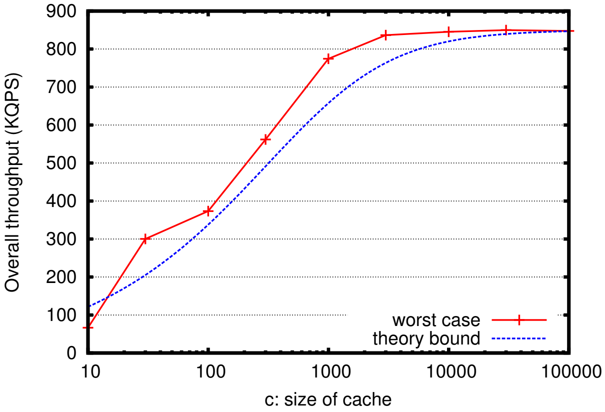


Figure 11: Adversarial throughput as the cache size increases. The "worst case" line shows the worst throughput achieved when testing ten different sizes of adversarial request sets.

In summary, the back-end throughput alone is low when x is small, as the load is not perfectly balanced as shown in Figure 9. In this region, however, a small cache is very efficient to prevent hotspots and ensure enough good performance. When x is 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 11 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).

6. DISCUSSION

Real-world clusters have additional functional requirements (e.g., negative caching) or behave in ways that violate some of the simplifying assumptions we made for our analysis (e.g., imperfect caching or handling queries with non-uniform processing costs). This section discusses how several of these complicating factors interact with small-fast-cache load balancing and how the caching may need to be adjusted to compensate, if possible. We conclude by discussing several opportunities for future enhancements to our work.

Multiple Front-End Caches Although a front-end cache can operate at a very high rate due to its small size, to scale, clusters will eventually require multiple front-end load balancers and caches to provide sufficient bandwidth and reliability. Our scalability results apply whether or not the system has multiple front-ends, but an interesting future question is whether such a system can use smaller individual caches on the front-end nodes because they handle less of the total load. The more challenging aspect of multiple front-ends, however, is cache consistency.

We believe that our solution is sufficient for weakly consistent services such as Dynamo, requiring only the addition of an efficient, asynchronous invalidation protocol between the caches. Strongly consistent services have two options: cache coherency or cache partitioning. The latter option doubles the amount of traffic that each front-end must handle by routing all queries for a particular key through its cache. This approach subtly weakens the load balance that can be provided by the cluster: in theory, if *all* requests arrived for only a single key, the cluster would be limited to the performance of a single front-end. However, provided that a front-end was, e.g., 100x faster than an individual back-end node, we suspect that this would not be too much of a problem in common practice. Nevertheless, it does not satisfy our desire to have the cluster provide provably high throughput guarantees.

The second approach of adding coherent caching introduces complexity, but is effective for read-mostly workloads. We must, however, defer to others solving the (perhaps impossible) general problem of supporting an intensive read-write mix to a single key at higher rate than can be handled by a single fast node.

Network Scaling We have assumed throughout this paper that the network itself has sufficient capacity to handle any traffic given to it. Emerging datacenter network designs such as fat-tree topologies may be able to make this assumption practical [5]. Absent them, particularly in the case of multiple front-end nodes, exploring the interaction of load balancing and network topological constraints seems a challenging problem for future work.

Imperfect Caching Policies Real-world caching policies are imperfect, particularly under an adversarial model. Fortunately, our results still apply even if the cache can be gamed, by using a slightly larger cache. For example, when the cache uses the least recently used (LRU) eviction policy, cycling $c + 1$ distinct keys will make every request miss the cache. However, much like our existing analysis, such behavior forces the adversary to spread its queries over $c + 1$ different keys. The difference from our analysis is that the previous adversarial strategy picked x^* keys to query, but incurred no benefit from the first c of these. With a predictable LRU cache, the first c also add load to the back-end nodes. Slightly increasing the k factor of the cache (e.g., by 1) can account for the practical extra load imbalance from these keys; the theoretical bound from Eq. (11) remains unchanged (it is already slightly pessimistic).

The fact that an adversarial workload that can completely bypass the cache is still handled helps illustrate why a small fast cache is effective in providing load balance: it arises not because the cache is fast, but because it forces the workload to be sufficiently uniform that it can be handled by the back-end nodes.

Gaming the Partitioning Scheme If the mapping of keys to back-end nodes is revealed to the adversary, a true adversary could launch an attack that requests the keys which are stored in the same node to overload the node. If such an attack is actually a concern, e.g., for the operator of a shared cloud storage infrastructure, naive attacks against the consistent hashing mapping can be defeated by using a keyed hash. Even then, however, the mapping could be probed using timing analysis by requesting a pair of uncached keys simultaneously and correlating their response latencies. The adversary may discover what key pairs are likely to be stored in the same node because queries for the keys in the same node will be likely to have similar response times within a short time frame. This scheme, however, may require a large number of trials to obtain a suitable level of confidence because external factors, such as network delay and front-end load balancer delay, will greatly increase the variance in the query latency measurement. We assume that the system operator can detect and block such attempts.

Non-Uniform Processing Costs The cost of serving each query may differ as the system may support many different types of operations (e.g., read, write, delete, and so on). A simple way to apply our analysis is to provide one properly-sized cache for each type of operation so that each can meet its SLA, but a more sophisticated analysis may be able to help tighten the bound on the size actually required.

Integration with Other Load Balancing Techniques We believe that small-fast-cache load balancing is compatible with several popular load balancing techniques. By combining caching with the other load balancing techniques, we may be able to achieve even more effective load balancing. In particular, as we mentioned in Section 2, we hope to explore integrating our work with "power-of-two-choices" load balancing.

During the evaluation of our system, we observed frequent instances in which individual back-ends took longer to complete work than others. In the cases we debugged, these problems arose primarily due to non-deterministic thread scheduling or hardware differences. A few were due to Ethernet auto-negotiation errors and interrupt handling in the front-end, and others arose because our testbed uses a heterogeneous mix of SSD drives. Had we used spinning disk drives, the variance would undoubtedly have been even higher. Anecdotal evidence from Google suggests that such performance variation is the norm, not an outlier [13]. Load balancing strategies that can cope with unpredictable back-end performance variation seem, therefore, mandatory.

As we noted earlier, two-choices load balancing cannot handle the adversarial workloads we consider, but neither does our caching cope with unpredictable back-end performance variation. We believe that combining the two could further reduce the size of the front-end cache, but have not yet proved it: informally, two-choices load balancing reduces the maximum load imbalance on a bin in the balls-in-bins model from $\frac{\log n}{\log\log n}$ to $\frac{\log\log n}{\log d}$; we suspect, therefore, that combining the two would allow the front-end cache to operate using only $O(N \log\log N)$—or even $O(N)$—cache slots instead of $O(N \log N)$. Given that $\log\log N$ is less than or equal to 5 for any currently feasible cluster size, this would confer a substantial advantage for the system designer.

Small Fast Caches Instead of Additional Replication Systems such as GFS are reported to replicate popular data items more than is necessary for fault-tolerance, in order to improve scalability and avoid hotspots [18]. With the help of this small load balancing cache, data replication might be returned to its role of serving only for failure recovery, permitting a reduction in the amount of storage. Clearly, a cache for a hard-drive based workload such as GFS will not fit in L3 cache, but it may fit well in a fast solid-state drive. The three order-of-magnitude access speed difference between a fast SSD and a hard drive seek is similar to the gap between L3 and DRAM or SSDs, and we believe our design principles would apply equally well to this larger scenario.

Future Work: a High-Speed Implementation Path We believe that a path exists to implement extremely high-speed front-end caches using the emerging crop of many-core CPUs. Processors using a Tilera-like architecture (64 cores with a total of 5.6MB of L2 cache) can today perform moderately complex functions such as deep packet inspection on 15 gigabits/second of network traffic. Our results suggest that their available memory is sufficient to create an extremely fast, efficient cache for a fast cluster. We believe that similar results could be achievable by building upon recent success in building many-core key-value store on Tilera [9] and fast software routers by carefully optimizing the CPU processing path [16], using GPUs to scale network routing [20].

7. RELATED WORK

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 [24]. 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 [28] in massive farms of memcached [25] servers, and Google maintains its entire index in tens to hundreds of terabytes of DRAM [13].

Server-based caching [10] 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.

8. CONCLUSION

Load balancing is an important problem in many large-scale distributed systems, both to achieve high performance and to meet service-level objectives for throughput and latency. 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 proved 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.

ACKNOWLEDGMENTS

We extend our particular thanks to several of the members of Matt Adiletta's group at Intel—Doug Carrigan and Johan van de Groenendaal in particular, for making it possible to assemble the 85-node FAWN testbed used for this research. We also acknowledge helpful comments from our SOCC reviewers and from Mor Harchol-Balter, Anupam Gupta and Kanat Tangwongsan.

This work was supported by funding from National Science Foundation award CCF-0964474, Google, the Intel Science and Technology Center for Cloud Computing, and by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. Hyeontaek Lim is supported in part by the Korea Foundation for Advanced Studies.

REFERENCES

[1] Hadoop. `http://hadoop.apache.org/`, 2011.

[2] HBase. `http://hbase.apache.org/`, 2011.

[3] Intel Threading Building Blocks. `http://threadingbuildingblocks.org/`, 2011.

[4] Apache Thrift. `https://thrift.apache.org/`, 2011.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *Proc. ACM SIGCOMM*, Aug. 2008.

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

[7] I. Ari, B. Hong, E. Miller, S. Brandt, and D. 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, Oct. 2003.

[8] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proc. 9th USENIX OSDI*, Oct. 2010.

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. `http://gigaom2.files.wordpress.com/2011/07/facebook-tilera-whitepaper.pdf`.

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

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

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[13] J. 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. ACM, 2009.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, Dec. 2004.

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

[16] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[17] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. 9th USENIX OSDI*, Oct. 2010.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[19] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th USENIX OSDI*, Nov. 2000.

[20] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, Aug. 2010.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. EuroSys*, Mar. 2007.

[22] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. 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. ACM, 2010. ISBN 978-1-4503-0053-7.

[23] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. 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. ACM, 1997.

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

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

[26] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[27] O. O'Malley and A. Murthy. Winning a 60 Second Dash with a Yellow Elephant. `http://sortbenchmark.org/Yahoo2009.pdf`, Apr. 2009.

[28] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. In *Operating Systems Review*, volume 43, pages 92–105, Jan. 2010.

[29] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD*, 1988.

[30] M. Raab and A. 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. Springer-Verlag, 1998.

[31] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7:48–66, Feb. 1998. ISSN 1066-8888.

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, Aug. 2001.

APPENDIX A. PROOF OF THEOREM 1

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,

$$\mathbb{P}\{\text{case1}\} \geq \mathbb{P}\{\text{case2}\}. \tag{13}$$

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

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

APPENDIX B. DERIVATION OF EQ. (8)

Eq. (8) 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. (8), we have the maximizer of Eq. (8) to be

$$x^* = 1 + 2(c-1)\left(1 + \frac{1}{\sqrt{1 + \frac{2\alpha^2 n \log n}{c-1}} - 1}\right). \tag{14}$$

APPENDIX C. ANALYSIS OF HETEROGENEOUS NODE CAPACITIES

Assume there are still n physical nodes as before and each hosts v virtual nodes on average. Therefore, there are in total $n' = vn$ virtual nodes.

Applying Eq. (9), the highest load among all virtual nodes is bounded by:

$$\frac{1 + \sqrt{1 + 2\alpha^2 \frac{n' \log n'}{c-1}}}{2} \cdot \frac{R}{n'}, \tag{15}$$

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

$$\frac{\left(\frac{1+\sqrt{1+2\alpha^2 \frac{n' \log n'}{c-1}}}{2} \cdot \frac{R}{n'}\right) z}{\frac{z}{n'} R} = \frac{1 + \sqrt{1 + 2\alpha^2 \frac{n' \log n'}{c-1}}}{2}, \tag{16}$$

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 :

$$c = (v + v \log_n v) k \cdot n \log n + 1, \tag{17}$$

the maximum load of any physical node is still bounded by

$$\frac{1}{2}\left(1 + \sqrt{1 + \frac{2\alpha^2}{k}}\right). \tag{18}$$