Distributed Load Balancing in the Face of Reappearance Dependencies

Kunal Agrawal
Washington University in St. Louis
St. Louis, Missouri, USA
kunal@wustl.edu

William Kuszmaull
Harvard University
Cambridge, Massachusetts, USA
william.kuszmaul@gmail.com

Zhe Wang
FoundationDB
Cupertino, California, USA
zhewang.kakaiu@gmail.com

Jinhao Zhao
Washington University in St. Louis
St. Louis, Missouri, USA
jinhaoz@wustl.edu

ABSTRACT

We consider the problem of load-balancing on distributed databases. We assume that data is divided into chunks and each chunk can be replicated on a constant number d of servers. When a request arrives, it is routed to one of the servers that contains the relevant chunk. Each server may store outstanding requests in a bounded queue and requests may be rejected if the queue is full. The goal is to design strategies for data distribution and request routing that minimize both the rejection rate and the average request latency.

What makes this problem technically difficult is reappearance dependencies: if a chunk x is accessed at multiple different time steps, then the set of d servers that it can be routed to is *the same* each time it is accessed. This is a substantial departure from classical balls-and-bins settings where each ball arrival introduces *fresh* randomness into the system.

We show that, with new algorithmic and analytical approaches, it is possible to overcome reappearance dependencies and construct algorithms with optimal rejection rate, latency, and queue size.

CCS CONCEPTS

• **Theory of computation** → *Distributed algorithms*; **Online algorithms**.

KEYWORDS

Distributed key-value stores; Balls into bins; Power of d choices; Load balancing

ACM Reference Format:
Kunal Agrawal, William Kuszmaull, Zhe Wang, and Jinhao Zhao. 2024. Distributed Load Balancing in the Face of Reappearance Dependencies. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3626183.3659968

SPAA '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3659968

1 INTRODUCTION

Distributed databases are widely deployed in today's systems in cloud applications and file systems. In these systems, a large amount of data is distributed across many servers. When a client requests a particular data item, the request is routed by a ***load-balancing algorithm*** to (one of the) server(s) where the item is stored. Each server has a queue to store outstanding requests, and is able to process $g = O(1)$ requests from the queue per time step. If the size of the queue ever exceeds some fixed queue length q, then the request at the end of the queue is rejected.

The goal of the load-balancing algorithm is to simultaneously achieve ***high throughput*** and ***low latency***. High throughput means that only a very small fraction of requests are rejected. Low latency means clients don't have to wait for a long time for their request to be processed (i.e., requests do not spend many time steps in a queue). Most prior theoretical work on this topic makes stochastic assumptions on the arrival pattern of requests [10, 15, 17, 21, 22, 25, 27, 28, 35].

In this paper, we will analyze this problem under adversarial assumptions. Suppose there are m servers and data is cut up into n (immutable) chunks—each chunk is replicated across $d = O(1)$ different (typically random) servers.[1] At each time step, up to m requests can arrive from the clients, each to a unique chunk. The client requests are generated by an oblivious adversary, who knows the load balancing algorithm, and who does not know the random bits used by the load-balancing and replication algorithms. In order to handle the oblivious adversary, the obvious approach is to distribute the chunks to servers in some random fashion. Under these circumstances, we would like to design a strategy for assigning chunks to servers that have an average rejection rate of at most $1/\text{poly}\, m$ (for a polynomial of our choice), average latency $O(1)$, and a worst-case latency of at most $\text{polylog}\, m$.

What makes this setting interesting is the interactions between different time steps. Consider, for example, the case of $d = 1$ (no replication), and consider the workload in which the same set S of m items is accessed on every time step. At first, this may seem fine—most items are assigned to servers that receive only $O(1)$ other requests. But, over time, the correlations between time steps cause a problem: the servers that receive more than g (processing rate) requests on time step 1 *also* receive more than g requests on

[1] Each chunk contains multiple data items.

time step 2, time step 3, and so on. These oversubscribed servers, which represent a constant fraction of all servers, will quickly fill up their queues (no matter how large the queue-size is!). In the steady state, a constant fraction of requests will end up being rejected. This simple argument, which was formalized in recent work by [34], means that it is impossible to achieve a rejection rate of $o(1)$ in the $d = 1$ case.

What is not clear, however, is whether it may be possible to achieve strong guarantees using replication $d > 1$. Here, there is a good reason to be optimistic: for many allocation problems, there are ***power-of-two-choices*** phenomena [3, 6, 8, 9, 11, 12, 15, 25, 30, 33], where having even just a few choices for where to place/route/store objects yields significantly better results than having only a single choice. At the same time, there is also a reason to be pessimistic: these same allocation results often *do not* extend to adversarial settings, where the same object/request can be inserted/accessed multiple times by an oblivious adversary (we will discuss this further in a moment).

Relationship to balls and bins, and why oblivious adversaries are hard. Before we dive into our results, it is helpful to understand the relationship between our problem and the problem of classical balls-and-bins load balancing [1, 25]. We can think of the m queues in our problem as ***bins*** that collect ***balls*** (i.e., client requests that have been routed to that server). Each ball can be identified by the chunk that is accessing: if a ball accesses some chunk i, then use $h_1(i), h_2(i), \ldots, h_d(i)$ to denote the set of bins (i.e., servers) where the ball can go. On each time step, a set of $O(m)$ distinct balls are added to the system and routed to bins via the load-balancing algorithm; and then, after this occurs, each bin deletes up to $g = O(1)$ balls from its contents. The main goal (achieving a low rejection rate) translates to making sure that, at any given moment, none of the bins contain more than, say, polylog m (or, ideally even fewer) balls.

There is a significant technical difference, however, between the above balls-and-bins problem and others that are commonly studied in the literature [3, 6, 8, 9, 11, 12, 15, 25, 30, 33]. In our problem, if the same ball (i.e., chunk) i is accessed in multiple time steps, then the set of random servers $h_1(i), h_2(i), \ldots, h_d(i)$ that it is capable of being routed to will be the *same* each time it is accessed. This creates unfortunate ***reappearance dependencies***: when a ball i is accessed, we cannot assume that the state of the queues is independent of the ball's random bits (i.e., of $h_1(i), h_2(i), \ldots, h_d(i)$), because those random bits may have *already affected* the state of the system in the past. In fact, this is not merely an analytical issue—in the case of $d = 1$, it was precisely what led to the impossibility result of [34].

The problem of reappearance dependencies has also made a notable appearance in recent work on the classical balls-and-bins problem [4–6]. Consider the task of inserting/deleting balls into m bins, with up to $k \gg m$ balls present at a time, and where each ball has 2 bins that it can choose from.[2] A celebrated result by Berenbrink, Czumaj, Steger, and Vöcking [9] says that, in the insertion-only setting, it is possible to assign balls to bins such that all m bins have almost exactly equal loads. Although the balls-and-bins result *does* extend to stochastic settings where balls are inserted/deleted at random, it has recently been shown to provably fail in settings where balls can be inserted, deleted, and reinserted (after being deleted) by an oblivious adversary (and where $k \gg m$). This quite surprising failure result, which was shown by Bansal and Kuszmaul at FOCS'22 [5], ends up being due to reappearance dependencies: because balls can be inserted, deleted, and later reinserted, the random bits for a ball (when it is reinserted) will have already impacted the state of the system in the past (when it was previously inserted). These seemingly minor dependencies turn out to enable adversarial attacks (even by an oblivious adversary) that cripple the classical power-of-two-choices result.

Thus, it is not clear, *a priori*, whether we should expect a positive or negative result in our setting. On one hand, Bansal and Kuszmaul's impossibility result does not appear to extend to our problem. On the other hand, the reappearance dependencies in our problem are in some ways even *more severe* than in the classical one. (For example, in the $d = 1$ case of classical balls-and-bins, reappearance dependencies do not affect the expected maximum load across the bins.) The question of whether these dependencies can be overcome to achieve positive results is the main topic of this paper.

Our results: Handling reappearance dependencies with algorithmic and analytical techniques. We begin with two positive results. The first result (Section 3) considers the ***greedy algorithm***, which simply routes each client request i to the least-loaded server out of its d choices $h_1(i), h_2(i), \ldots, h_d(i)$. We prove that there exist positive constants g, d such that this strategy offers the following guarantee: assuming that each queue has capacity $q = \log m + 1$, that each server processes g requests per time step, and that each chunk is stored in d random locations, the greedy load-balancing algorithm achieves an expected rejection rate of $O(1/\text{poly}\, m)$ and an expected average latency of $O(1)$. Our analysis of the greedy algorithm can be viewed as a new variation on the classical *layered-induction* technique [3]; the difference is that, in order to handle reappearance dependencies, we restructure the argument in such a way that we are able to apply, in one part of the argument, a very broad union bound—so broad, that we are able to wipe away any possible effects of reappearance dependencies.

The second result (Section 4) introduces a new load-balancing algorithm that we call ***delayed cuckoo routing***. The delayed cuckoo routing algorithm is somewhat more involved than its greedy counterpart, and requires that the load balancing algorithm maintain an additional data structure beyond the queues. The benefit of the algorithm, however, is that it offers an even stronger guarantee. Using 2-way replication, constant processing rate g, and queues of size just $q = O(\log \log m)$, the algorithm is once again able to achieve a rejection rate is $1/\text{poly}\, m$ and expected average latency $O(1)$.

Part of what makes delayed cuckoo routing interesting is that the guarantees it achieves are provably optimal. We show in Section 5 that any load balancing algorithm using $d = O(1)$ and $g = O(1)$ and achieving expected rejection rate $O(1/m)$ must use queues of size $q = \Omega(\log \log m)$. We also show that any load balancing algorithm with $d = O(1)$ and $g = O(1)$ must have an expected rejection rate

[2] Typically, in the ball-and-bins literature, m is used to denote the number of balls and n is typically used for the number of bins. However, for notational consistency, we adopt m as the number of bins and k as the number of balls.

of at least $1/\text{poly}\, m$. This means that the rejection rates achieved by both of our algorithms are optimal.

Our final result offers a simple but intriguing impossibility result. Suppose that, within each time step, we wish to make routing decisions that are independent of the requests made in previous time steps. That is, we wish to handle the reappearance dependencies between time steps by simply making our routing decisions within a time step (i.e., which of the d choices to use for each request) random enough that they do not interfere with choices made in other time steps. We show that, somewhat surprisingly, such an algorithm is provably non-viable – even in the setting where the set of requests that are made is the same across time steps. This lower bound comes with a balls-and-bins interpretation that will likely be of independent interest.

2 PRELIMINARIES

In this section, we describe our model and optimization criteria.

Problem definition. We assume that all the data is divided into n chunks (each chunk potentially contains multiple keys) and stored on a total of m servers where each chunk is stored (replicated) on $d = O(1)$ servers. On every time step, up to m requests are generated by clients to unique chunks. When a client requests some data, this request can be routed to any server that has the corresponding chunk that stores the data. Each server has a FIFO queue of some bounded ***queue length*** q and requests are stored in this queue. On every time step, a server satisfies $g = O(1)$ requests, where g is known as the ***processing rate***. At any given moment, the number of unprocessed requests in a server's queue is referred to as the server's ***backlog***. The server must maintain a backlog of at most q (the maximum queue length), and in order to do this the server may sometimes choose to ***reject*** a request.

Given, d, g, m, q, there are three algorithmic knobs. The first knob is the choice of where each chunk's d replicants reside – our algorithms will assume that each replicant is simply assigned to a random server. The second and more important knob is the load-balancing algorithm that routes each request to a server. Note that any algorithm we design must be *online* (even within the m requests that are performed during a time step), which is to say that each time a request is made, it is routed immediately to a server without any knowledge of future requests. Finally, the third (and least important) knob is the choice of when to reject requests – in principle, a server may choose to reject a request even if the server's queue is not full. As we shall see, this can be helpful for handling rare failure events in which we need to "reset" the system.

Optimization criteria. There are two goals for the design of a good system. The first goal is to maximize throughput – in other words, to minimize the fraction of requests that are rejected, also known as the rejection rate. The second goal is to minimize both average and worst case latency. The maximum latency is nominally the length of the longest queue. The average latency, on the other hand, is proportional to the average backlog across the servers over time.

Formally, we define these objectives as follows.

Definition 2.1 (Rejection Rate). Given a sequence σ of requests arriving over time, $T_A(\sigma)$ denotes the number of requests that are accepted by algorithm A on sequence σ. The rejection rate is $(|\sigma| - T_A(\sigma))/|\sigma|$.

Our goal is to design systems that achieve expected rejection rate $O(1/\text{poly}\, m)$. (And, as we shall see in Section 5, this is optimal.)

Definition 2.2 (Latency). Given a sequence σ of requests arriving over time, $L_A(\sigma_i)$ denotes the number of time steps before this request is satisfied by the server. The ***maximum latency*** is $\max_i L_A(\sigma_i)$, and the ***average latency*** is $\sum_i L_A(\sigma_i)/|\sigma|$.

In our systems, our goal is to design systems where the maximum latency is logarithmic or sub-logarithmic in m and the average latency of $O(1)$. Our delayed cuckoo routing algorithm, in particular, will achieve $O(\log \log m)$ maximum latency, which will turn out to be optimal.

Basic observations. We conclude the section with some basic observations. First note that, if the processing rate g were less than 1 (say, $1 - \Omega(1)$), then it would be impossible to process requests at the rate that they arrive. Therefore, we are only interested in $g \geq 1$, and our goal will be to offer guarantees for $g = O(1)$. Also observe that the constraint on requests within a timestep – namely, that they access a *distinct* set of chunks – is necessary. If $\omega(1)$ requests could be made to a single chunk, and those requests were repeated on every time step, then the $d = O(1)$ servers that store that chunk would be forced to reject most of the requests. Finally, since prior work has already ruled out the possibility of deterministic solutions to our problem [34], we cannot hope for a rejection rate of 0. We can, however, hope for a rejection rate of $o(1)$, or more concretely, as we will achieve in later sections, $1/\text{poly}\, m$.

3 ANALYSIS OF GREEDY ALGORITHM

We now analyze the greedy algorithm, which routes each request to the queue with the least backlog out of the d queues that can handle the request. If a queue ever overflows, then it rejects all of its requests. Additionally, for some positive constant c, the queues reject all of their outstanding requests (clearing out the system) once every m^c steps.

The main result of the section will be the following:

Theorem 3.1. *For any positive constant $c > 0$, there exists $d, g = O(1)$ such that the following holds. Supposing replication d, server processing rate g, and queue length $q = \log m + 1$, the greedy algorithm has expected rejection rate $O(1/m^{c-1})$, maximum latency at most $O(\log m)$, and expected average latency at most $O(1)$.*

We shall assume throughout the section that d is a sufficiently large positive constant with respect to c and that g is a sufficiently large positive constant with respect to d.

In order to prove this theorem, we will argue that the distribution of the number of outstanding requests in the queues obeys a distribution — we will call this a ***safe*** distribution.

Definition 3.2 (Safe distribution of backlogs). Consider the number of requests in the queues of m servers. The distribution is said to be safe if for all $1 \leq j \leq \log m$, at most $\frac{m}{2^j}$ queues have more than j requests. Formally, let p_i be the number of servers with backlog exactly i. We know that $\sum_{i=0}^{\infty} p_i = m$. The balls are in a safe distribution if $\sum_{i=j+1}^{\infty} p_i \leq \frac{m}{2^j}$ for all j.

We will prove by induction that, with high probability, the distribution of queue backlogs is guaranteed to be in a safe distribution at any given moment.

As noted earlier, what makes this analysis difficult are *reappearance dependencies*. Consider a chunk x that is accessed at some time t, and let us refer to the servers where x's replicants are stored as x's ***hashes***. Because x may have also been accessed in the past (prior to time t), the current state of the system *cannot* be said to be independent of x's random hashes. Even if we assume that we are in a safe distribution (i.e., at most $m/2^j$ servers have backlog $\geq j$), the actual *set* S_j of servers that have backlog $\geq j$ for some j is a random variable that depends on x's hashes! Even if we can guarantee that $|S_j|$ is small, how can we guarantee that it is not adversarial against x (so that, for example, all of x's hashes end up being in S_j)?

A key insight in the analysis will be that, even if the set S_j is adversarial against some chunk x, it cannot *simultaneously* be adversarial for all (or even a large fraction) of the m requests that are made in a given time step. In fact, by considering all of these requests together, we will be able to apply a union bound over *all possible options* for S_j in order to argue that, even if S_j were determined adversarially, it would not prevent us from completing our analysis. This is how we will get around reappearance dependencies.

The main union bound in the analysis is captured by the following lemma.

Lemma 3.3. *Consider any set of $\frac{m}{g}$ chunks $H = \{x_1, x_2, \ldots, x_{m/g}\}$, and consider any $k \leq m/g$. We have with probability $1 - O(1/m^{3c+2})$ that every subset $H' \subseteq H$ of size k has the following property: the number of distinct servers that contain at least one replicant of at least one chunk in H' is greater than $4k$.*

Proof. Suppose that all of the replicants for H' are stored in some set S of $4k$ servers. There are $\binom{m}{4k}$ ways to choose the servers S and $\binom{m/g}{k}$ ways to choose the chunks of H'. Given the chunks and the servers, the probability that all the chunks have all their replications on the servers is

$$\left(\frac{4k}{m}\right)^{dk}.$$

Therefore, from a union bound, the probability that there exist H' and S that violate the lemma is at most

$$\binom{m}{4k}\binom{m/g}{k}\left(\frac{4k}{m}\right)^{dk},$$

which by the bound of binomial coefficients is at most

$$\frac{e^{4k}m^{4k}}{(4k)^{4k}} \cdot \frac{e^k(m/g)^k}{k^k} \cdot \frac{4^{dk}k^{dk}}{m^{dk}}$$

Using the fact that g is a sufficiently large positive constant with respect to d, we can bound the above expression by

$$\frac{1}{2^{\Omega(k)}} \cdot \left(\frac{m}{k}\right)^{5k} \cdot \left(\frac{k}{m}\right)^{dk}.$$

Using the fact that d is also a sufficiently large positive constant, we can further upper bound our expression by

$$\frac{1}{2^{\Omega(k)}} \cdot \left(\frac{k}{m}\right)^{dk/2}$$

If $k = \omega(\log m)$, then the first multiplicand is sub-polynomial, and otherwise the second multiplicand is at most $\tilde{O}(1/m)^{d/2} \leq O(1/m^{3c+2})$. □

We will now show that if the servers are in a safe distribution, then the safe distribution will be maintained over time with high probability. We will perform the analysis at the level of ***time sub-steps*** where, on each sub-step, we get m/g requests and each server consumes 1 request in each sub-step.

Lemma 3.4. *Suppose we start a sub-step t with a safe distribution. Then, at the end of the sub-step, we will still be in a safe distribution with probability $1 - O(1/m^{3c+1})$.*

Proof. Consider $k = \frac{m}{2^j}$. We prove by induction that, at the end of the sub-step, there are at most $\frac{m}{2^j}$ queues with more than j outstanding requests, from j equals 0 to $\log m$. The base case of $j = 0$ is trivial.

Each step of the induction will apply Lemma 3.3, adding an additional $O(1/m^{3c+2})$ to our total probability of failure. thus, over the $O(\log m)$ induction steps, the total failure probability will be at most $O(1/m^{3c+1})$.

Let the servers with more than i requests be $S_i^{(t)}$ before sub-step, and be $S_i^{(t+1)}$ after. We know by assumption $|S_j^{(t)}| \leq \frac{m}{2^j}$ because the queues are initially in a safe distribution. Moreover, by the inductive hypothesis, $|S_{j-1}^{(t+1)}| \leq \frac{m}{2^{j-1}}$ (with high probability).

For a server $w \in S_j^{(t+1)}$, consider whether w has requests among the m/g new-coming requests.

Case 1: w has at least one new-coming request. Let the last request (in time) that comes to w be x. We claim that x must have all its d replications in $S_{j-1}^{(t+1)}$. If not so, it has a replication w' with queue length at most $j-2$ at time $t+1$ (so when x arrives, the length of the queue on w' is at most $j-2$), and yet it goes to w which has queue length j at time $t+1$ (so when x arrives, the length of the queue on w is at least $j-1$). This means x should choose w' instead of w, which is contradictory. According to Lemma 3.3, with probability $1 - O(1/m^{3c+2})$, the number of requests (corresponding to chunks) of this case is no more than $\frac{1}{4} \cdot |S_{j-1}^{(t+1)}| \leq \frac{m}{2^{j+1}}$. Thus the number of servers w in this case is also at most $\frac{1}{4} \cdot |S_{j-1}^{(t+1)}| \leq \frac{m}{2^{j+1}}$.

Case 2: w has no new-coming requests. This means $w \in S_{j+1}^{(t)}$ because the server consumes a request. We know by induction that $|S_{j+1}^{(t)}| \leq \frac{m}{2^{j+1}}$, so the number of servers w in this case is at most $\frac{m}{2^{j+1}}$.

Each server in $S_j^{(t+1)}$ is in one the two cases. Thus (unless the inductive hypothesis fails), we have with high probability that $|S_j^{(t+1)}| \leq \frac{m}{2^{j+1}} + \frac{m}{2^{j+1}} = \frac{m}{2^j}$, which completes the induction step. Since there are $O(\log m)$ induction steps, and since each has at most $O(1/m^{3c+2})$ probability of failing, the entire lemma holds with probability $1 - O(\log m/m^{3c+2}) \geq 1 - O(1/m^{3c+1})$.

□

Finally, we can complete the proof of Theorem 3.1.

Proof of Theorem 3.1. The worst-case latency is trivially at most the queue size of $\log m + 1$. Therefore we focus on analyzing rejection rate and average latency.

Recall that the queues are flushed every m^c time steps. Consider the time interval between two flushes, and call it a ***phase*** – it suffices to prove that our desired bounds hold within each phase.[3]

By Lemma 3.4, we have probability at least $1 - O(m^c/m^{3c+1}) \geq 1 - O(1/m^{2c+1})$ that every sub-step in the phase ends in a safe distribution. In the $1/m^{2c+1}$-probability event that there is ever a non-safe distribution during the phase, we can feel free to simply think of all m^{c+1} of the requests in the phase as having been rejected – this still only adds $1/m^c$ to the expected rejection rate during the phase. On the other hand, assuming that every sub-step in the phase ends in a safe distribution, then the only rejections that occur during the phase are in the final step where all queues are flushed. Since the queues are in a safe distribution when they are flushed, the total number of flushed requests is $O(m)$ – these flushes therefore contribute at most $O(1/m^{c-1})$ to the overall rejection rate. Putting the pieces together, the total expected rejection rate during the phase is at most $O(1/m^{c-1})$.

Finally, to analyze average latency, it suffices to show that the average queue length is at most $O(1)$. Since, at any given moment, the probability of being in a safe distribution is very high ($\geq 1 - O(1/m^{2c+1})$), the contribution of steps that are not in safe distributions is negligible. On the other hand, for steps that are in safe distributions, the average queue length is at most the number of requests present in queues (at most $m + m/2 + m/4 + \cdots \leq 2m$) divided by the number of queues m. Thus, the average queue-length overall, at the end of each time sub-step, is at most $2 + o(1)$. This, in turn, implies an expected average latency of $O(1)$. □

4 REDUCING MAXIMUM LATENCY VIA DELAYED CUCKOO ROUTING

We saw in the previous section that the simple greedy algorithm is able to provide rejection rate of $O(1/\text{poly}\, m)$, expected average latency $O(1)$ and maximum latency of $O(\log m)$ using queues of size $\Theta(\log m)$. We now consider whether we can design a strategy that provides a better bound on maximum latency (and thus also queue size) by leveraging past information about requests. As we shall discuss in Section 5, classical balls-and-bins lower bounds prohibit the possibility of using queues of size $o(\log \log m)$. Thus, our target is to design a strategy that can use queues of size $O(\log \log m)$.

In this section, we will describe a strategy called ***delayed cuckoo routing*** that uses cuckoo hashing as a critical subroutine. The intuition behind this strategy can be understood by considering two extreme cases. First, say that the requested chunks are not repeated frequently: that is, across nearby time steps, the clients tend to request different chunks. In this case, reappearance dependencies are not a problem, and greedy algorithm already guarantees a maximum latency of $O(\log \log m)$. This suggests that, intuitively, the worrying case is when the same m chunks are requested over and over again. However, in this case, we can use cuckoo hashing to pre-compute where these requests should be routed, guaranteeing that each server only receives a constant number of requests per time step. Since, in general, we will not be in either of these extremes, we will design a strategy that combines the two to handle any workload.

[3]These periodic flushes allow us to bound the impact of low probability events which might cause us to leave the safe distribution since these can only break the system until the next flush.

Background on Cuckoo Hashing

Cuckoo hashing, which was first introduced by Pagh and Rodler in 2001 [29], is a technique for assigning a set of up to, say, $m/3$ (or, more generally, $m/2 - \Omega(m)$) items to m positions so that each position receives at most one item, and such that each item x is in one of two random positions $h_1(x) \in [m]$ and $h_2(x) \in [m]$ that it hashes to. Although classical Cuckoo hashing has a $\Theta(1/m)$ failure probability, the probability can be reduced to $1/\text{poly}\, m$ by allowing for $O(1)$ items to be excluded from the allocation [18] (this is known as cuckoo hashing *with a stash*). If one allocates these $O(1)$ items arbitrarily, then every position still receives at most $O(1)$ items – this leads to the following formulation of Krisch et al.'s [18] analysis of cuckoo hashing with a stash:

Theorem 4.1 (Cuckoo Hashing with a Stash [18]). *Consider a set S of $m/3$ items, and let h_1, h_2 be fully random hash functions mapping each $x \in S$ to a random element of $\{1, 2, \ldots, m\}$. Let $c > 0$ be an arbitrary positive constant. With probability $1 - O(1/m^c)$, it is possible to assign the elements of S to positions in $[m]$ such that each position receives at most $O(1)$ elements and such that each element $x \in S$ is in one of positions $h_1(x)$ or $h_2(x)$.*

Cuckoo hashing can also be implemented as an online algorithm, in which items are moved around over time so that, at any given moment, there is at most one item in each position. However, this online variant of Cuckoo hashing will not be very helpful to us, since when a request arrives, we must make an irrevocable and immediate decision about which server to route it to – i.e., we cannot change our routing decisions after the fact in the same way that cuckoo hashing moves around items.

Nonetheless, if we *did* know the full set of items that were going to arrive on a given time step, we could use Cuckoo hashing to perform very good load balancing on the m servers. This is captured by the following lemma, which follows immediately from applying Theorem 4.1 three times.

Lemma 4.2. *Say we have replication $d = 2$, and let $c > 0$ be a positive constant. Given m requests to unique chunks, it is possible to assign them to servers so that, with probability at least $1 - O(1/m^c)$, every server receives at most $O(1)$ requests.*

4.1 Delayed Cuckoo routing

We now describe the ***delayed cuckoo routing***, which is a load-balancing algorithm that uses replication $d = 2$, processing rate $g = O(1)$, and queues of size $q = \Theta(\log \log m)$. For each chunk x, we will use $h_1(x)$ and $h_2(x)$ to indicate the two random servers where the chunk is stored.

For each time step i, let S_i denote the set of chunks that are accessed in that step, and let T_i denote the assignment that Lemma 4.2 produces on S_i. For each chunk $x \in S_i$, we will use $T_i(x)$ to denote the server in $\{1, 2, \ldots, m\}$ that T_i assigns x to.

We cannot use T_t *during* time step t because we cannot construct T_t until after we know S_t. However, we can use T_t to help us make good routing decisions in the future.

The delayed cuckoo routing algorithm runs in phases of $\log\log m$ time steps. During each phase j, each server i maintains two queues Q_i and P_i of lengths $O(\log\log m)$, each of which processes $g/4$ items per time step. When a chunk x is requested at some time t, if it is the first time that the chunk has been requested during phase j, then the request is placed in whichever queue $Q_{h_1(x)}$ or $Q_{h_2(x)}$ has fewer items (the request is rejected if both queues are at capacity). If, on the other hand, the chunk x has already been requested in the past during phase j, then we consider the most recent time $t' < t$ that the chunk x was requested, and we place the request in queue $P_{T_{t'}(x)}$. (If $T_{t'}$ experienced as failure event, which by Lemma 4.2 occurs with probability $1 - O(1/m^c)$, then the request for x is rejected.)

Additionally, at the beginning of a phase $j > 1$, there may still be requests queued up in the servers from the previous phase $j - 1$. We move these requests to queues P'_i and Q'_i (so now there are four queues in each server), and we also process $g/4$ requests from each P'_i and Q'_i on each step. Since $|P'_i|, |Q'_i| \leq O(\log\log m)$ deterministically, they are guaranteed to be empty by the end of phase j (so long as g is a sufficiently large positive constant).

As a minor remark, the reader may notice that each server i is no longer maintaining just a single FIFO queue, but instead (up to) four FIFO queues. This distinction is only to simplify the analysis, however. Indeed, if one were to process the items in $P_i \cup Q_i \cup P'_i \cup Q'_i$ in true FIFO order, it would not change total number of requests queued at server i at any given moment, the rejection rate, or the average latency achieved.

In the rest of the section, we will prove the following theorem:

Theorem 4.3. *Let $c > 0$ be an arbitrary positive constant, let $g = O(1)$ be a sufficiently large positive constant, and let $q = \Theta(\log\log m)$ be a sufficiently large multiple of $\log\log m$. Supposing replication $d = 2$, server processing rate g, and queue length q, the delayed cuckoo routing strategy has expected rejection rate $O(1/m^c)$, maximum latency $O(\log\log m)$, and expected average latency $O(1)$.*

The bound on maximum latency is immediate from the queue length $q = O(\log\log m)$. We will split the rest of the section into two parts, one analyzing expected rejection rate, and one analyzing expected average latency.

4.2 Bounding the rejection rate

We will now argue that the expected rejection rate of delayed cuckoo routing is $O(1/m^c)$.

Consider a request r to some chunk x at some time t. We begin by considering the case where this is the first access to x in the current phase. In this case, there are no reappearance dependencies to worry about, so we can use standard balls-and-bins arguments to bound the probability that r is rejected.

Lemma 4.4. *If r is the first request to x in the current phase, then the probability of r being rejected is $O(1/m^c)$.*

Proof. Let us imagine for a moment that each queue Q_i has unbounded length. It suffices to show that, with probability at least $1 - O(1/m^c)$, the number of unprocessed requests in each Q_i, at any given moment, is at most $O(\log\log m)$.

Recall that, when a request to a chunk x is placed into a queue Q_i, it is placed into whichever of $Q_{h_1(x)}$ or $Q_{h_2(x)}$ has fewer unprocessed requests. Define $q_{i,t}$ to be the number of requests in queue Q_i at time t.

Now consider an alternative reality in which each chunk x (the first time it is accessed during the phase) is assigned to whichever of $Q_{h_1(x)}$ or $Q_{h_2(x)}$ has received fewer *total* requests during the phase. And let $q'_{i,t}$ be the *total* number of requests that queue Q_i receives during the phase, in this alternative reality.

Note that the $q'_{i,t}$ values dominate the $q_{i,t}$ values. Indeed, it is easy to see by induction that $q_{i,t} \leq q'_{i,t}$ for all i, t. Thus, if we wish to prove an upper bound on $q_{i,t}$, it suffices to prove an upperbound on $q'_{i,t}$.

However, the task of bounding $q'_{i,t}$ is equivalent to a standard balls-and-bins analysis. A classical result by Berenbrink et al. [9] says that, if mh balls are placed into m bins, and each ball is placed in the less-loaded of two random bins, then the fullest bin will have $mh + O(\log\log m)$ balls with probability $1 - 1/\text{poly}\, m$ (for a polynomial of our choice). In our case, $h = O(\log\log m)$, so we have that $q'_{i,t} \leq O(\log\log m)$ with high probability. This completes the proof. □

Next we consider the case where x has already been accessed in the past, during the current phase. Let $t' < t$ be the most recent time in the past that x was accessed. Recall that, in this case, x is placed into queue $P_{T_{t'}(x)}$ – if the cuckoo hash table $T_{t'}$ experiences a failure (which by Lemma 4.2 occurs with probability $O(1/m^c)$), then request r is rejected. Otherwise, we claim that queue $P_{T_{t'}(x)}$ is guaranteed (deterministically!) to have room for request r without overflowing.

Lemma 4.5. *With probability one, each queue P_i has at most $O(\log\log m)$ requests routed to it during a given phase. Thus, the queue will never exceed its capacity of $O(\log\log m)$.*

Proof. Let $[t_1, t_2]$ be the sequence of $\log\log m$ time steps that constitute the current phase. The total number of requests that are assigned to P_i during the phase is at most

$$\sum_{t \in [t_1, t_2]} \sum_{x \in S_t} \mathbb{I}[T_t(x) = i].$$

By Lemma 4.2, this is at most

$$\sum_{t \in [t_1, t_2]} O(1) \leq O(\log\log m),$$

as desired. □

Lemma 4.5 tells us that, in the case where x has already been accessed before in the phase, the only failure mode for x (i.e., the only case where x can be rejected) is if $T_{t'}$ itself fails. Since this happens with probability $O(1/m^c)$, we have the following lemma:

Lemma 4.6. *If a request r is to a chunk x that has already been accessed previously in the same phase, then the probability of the request being rejected is $O(1/m^c)$.*

Combining together Lemmas 4.4 and 4.6, we arrive at a bound on the expected overall rejection rate:

Proposition 4.7. *The expected rejection rate of delayed cuckoo routing is $O(1/m^c)$.*

4.3 Analysis of Average Latency

Finally, we bound the expected average latency of delayed cuckoo routing. For this, it suffices to bound the expected average latency for the requests within a given phase. We remark that the analysis of average latency in this section requires a much more subtle probabilistic argument than did the analysis for the greedy algorithm.

The main component of the analysis is the following technical lemma.

Lemma 4.8. *Let $X_j^{(t)}$ be the number of requests that come to the P_j on time step t. Then for any time interval $I = [T, T+\ell)$ within a given phase, we have $\Pr[\sum_{t=T}^{T+\ell-1} X_j^{(t)} \geq g\ell/4] \leq e^{-\ell}$.*

Proof. Say $Z_{i,j}$ is an indicator random variable which is 1 if request i goes to server j. Then the sum $Y_j := \sum_{t=T}^{T+l-1} X_j^{(t)}$ is a sum of $Z_{i,j}$'s for lm different values of i. Since $\Pr[Z_{i,j} = 1]$ is trivially $1/m$, it is easy to see that $\mathbb{E}[Y_j] = \ell$. However, for a given j, the random variables $\{Z_{i,j}\}$ are (highly) not independent. In particular, if the same chunk is requested over and over again, the requests to it can only go to one of two queues.

Since the requests under consideration end up in $\{P_j\}$ queues, there must be a prior request q' to the same chunk at some earlier time within the same phase. If the most recent such request takes place in time interval $I = [T, T+\ell)$, we call q a ***fresh*** request; otherwise (if q' takes place before time T), we call q a ***stale*** request. Note that the definition of *fresh* and *stale* is dependent on T and ℓ.

Each of the ℓm requests is either *fresh* or *stale*. First consider fresh requests. Note that the fresh requests are routed according to their cuckoo hashing results from a prior step which was also within the interval under consideration. According to Theorem 4.2, at most $O(1)$ requests can have their cuckoo hash result be a particular server at any time step. Therefore, at most $O(\ell)$ fresh requests can be routed to a particular queue P_j within an interval of length l.

The stale requests, on the other hand, are all to *different chunks with one another*. Thus, each stale request *independently* has probability at most $2/m$ of being placed in queue P_j (indeed, it has probability at most $2/m$ that the chunk x being accessed satisfies $j \in \{h_1(x), h_2(x)\}$). Suppose there are w stale requests, and $H_1, \ldots, H_w$ are the 0/1 random variable representing whether each stale request hashes to server j (i.e., $j \in \{h_1(x), h_2(x)\}$). From a Chernoff Bound we know that

$$\Pr\left[\sum_{i=1}^{w} H_i \geq 6\ell\right] \leq (e^{\delta}/(\delta+1)^{\delta+1})^{\ell} \leq e^{-\ell}.$$

where $\delta = 3\ell m/w - 1 \geq 2$.

Summarizing, for the fresh requests, the summation of $Z_{i,j}$s is less or equal than $O(\ell)$; for the stale requests, the $Z_{i,j}$s sum to at most $\sum_{i=1}^{w} H_i$, which is at most 6ℓ with probability $1-e^{-t}$. Therefore, using the fact that g is sufficiently large as a positive constant, we have

$$\Pr\left[\sum_{t=T}^{T+\ell-1} X_j^{(t)} \geq g\ell/4\right] \leq e^{-\ell}.$$

□

Lemma 4.8 shows that for any queue P_j, in any time period, it is not likely to have too many requests. We use this to obtain a bound on expected average latency.

Proposition 4.9. *The expected average latency of the delayed cuckoo hashing algorithm is $O(1)$.*

Proof. For requests routed to servers Q_j, the expected average latency is at most $O(1)$ by the same argument of Theorem 3.1.

Thus it suffices to analyze requests routed to servers P_j. We will show that each such request has probability at most $e^{-\Omega(k)}$ of having latency at least k – thus, the expected latency of such a request is $O(1)$, as desired.

Consider a request that is placed in server P_j at time t and that has latency at least some value k. Let $t-\ell-1$ be the most recent time at which server P_j was empty. Then, during time interval $[t-\ell, t)$, P_j processed $g\ell/4$ requests and still ended up with k unprocessed requests. This means that, during time interval $[t-\ell, t)$, the total number of requests that were sent to P_j was at least $g\ell/4 + k$. It follows that there exists some $\ell' = \max(\ell, 4k/g) \geq \Omega(k)$ such that, during time interval $[t-\ell', t)$, at least $g\ell'/4$ requests were routed to P_j. By Lemma 4.8 and the union bound, the probability of any such ℓ' existing is, for a given queue P_j, at most $\sum_{\ell' \geq \Omega(k)} e^{-\ell'} = e^{-\Omega(k)}$.

In order for a given request to some chunk x to be routed to a P_j-queue and have latency at least k, at least one of $P_{h_1(x)}$ or $P_{h_2(x)}$ must experience this $e^{-\Omega(k)}$-probability event. Thus, for any given request and any given k, the probability that the request experiences latency at least k in some server P_j is at most $e^{-\Omega(k)}$. This completes the proof. □

5 LOWER BOUNDS

In this section, we prove three lower bounds. The first bound, which is essentially a reinterpretation of Vöcking's now classical power-of-d-choices lower bound [33], shows that queues of length $\Omega(\log\log m)$ are necessary in any system that wishes to reject $O(1)$ requests, on average, per time step. This means that delayed cuckoo routing is optimal for both queue length and maximum latency.

Theorem 5.1. *With constant replication $d = O(1)$ and constant server capacity $g = O(1)$, if the expected rejection rate is $O(1/m)$, the length of the queue must be $\Omega(\log\log m)$.*

Proof. Consider a single time step in which m requests are made to independently random chunks (from a sufficiently large universe of chunks). From Theorem 2 of [33], any online algorithm for placing m balls into m bins, where each ball has $d = O(1)$ bin choices drawn from some distribution over $[m]^d$, we have with probability $1-o(1)$ that there exists a bin that receives $\Omega(\log\log m)$ balls. This means that, in our problem, some server will receive at least $\Omega(\log\log m)$ requests in the first time step. If the length of the queue is $o(\log\log m)$, there will be $\Omega(\log\log m)$ requests that are rejected. This implies a rejection rate of $\Omega((\log\log m)/m)$, which is a contradiction. □

Next we show that, in general, one cannot hope for a better-than-polynomial rejection rate. This captures a sense in which both of our algorithms are optimal.

Theorem 5.2. *With constant replication $d = O(1)$ and constant server processing rate $g = O(1)$, the expected rejection rate is at least $1/m^{O(1)}$.*

Proof. Suppose that the number of distinct chunks is $n \geq m^{\omega(1)}$ and consider a set S of m i.i.d. chunks drawn uniformly at random. Because $n \geq m^{\omega(1)}$, we have with probability $1 - 1/n^{\omega(1)}$ that the m i.i.d. chunks are distinct (if they are not, say that a ***sampling error*** has occurred and abort the construction). Consider the event E that the first $gd + 1$ chunks have completely identical positions for their replications. The probability of this happening is at least $1/m^{gd} = 1/m^{O(1)}$ (since the chunks are i.i.d., and so their d choices are i.i.d. from some distribution over $[m]^d$, and the distribution over $[m]^d$ with smallest collision probability is the uniform one). Since the probability of a sampling error is $1/m^{\omega(1)}$, the probability of the event E' that E occurs without any sampling errors is at least $1/m^{O(1)} - 1/m^{\omega(1)} \geq 1/m^{O(1)}$. If we condition on E', then on every time step, there are d positions that collectively consume at most gd requests but receive at least $\geq gd + 1$ requests. This implies that, conditioning on E', the rejection rate is at least $\Omega(1/m)$. Since E' occurs with probability $1/m^{O(1)}$, the expected rejection rate overall is at least $1/m^{O(1)+1} = 1/m^{O(1)}$.

□

Our final lower bound considers the possibility that a so-called time-step-isolated load-balancing strategy could achieve strong guarantees. A strategy is said to be ***time-step isolated*** if, with each time step, it makes its routing decisions based only on the requests made during that time step (and not based on information about which requests are in queues at the beginning of the time step, or more generally, information about what occurred in past time steps).

We show that, somewhat surprisingly, time-step isolated strategies fail even in the setting where the same sequence of m chunks is accessed over and over again; and, in fact, even in the setting where queue length is infinite.

Lemma 5.3. *Consider any time-step isolated strategy with $d = O(1)$. Let σ be a sequence of m random chunks (from some sufficiently large universe) that is used as the request sequence on time step. With probability $1 - o(1)$ (where the randomness comes from the choice of which servers store which chunks), there exists some bin $j \in [m]$ that the time-step isolated strategy sends $\Omega(\log \log m)$ balls to, on average, per time step.*

We remark that Lemma 5.3 also has a natural alternative interpretation. Suppose that every day, m requests are made and are routed to servers using d random choices for each request; and suppose that our goal over time is to route the requests in such a way that there is no *single* server that receives $\omega(1)$ *average* load per day (note that we don't care about worst-case load on a particular day at all here, we just want to assign loads fairly *over time*). Lemma 5.3 says that, if the routing on each day is performed in an online fashion using only information from that day, then this seemingly simple load-balancing problem is impossible.

Proof. Suppose for contradiction that every in $j \in [m]$ receives an average load of $o(\log \log m)$ requests per day. Then we will prove the existence of an online d-choice balls-and-bins strategy $\overline{\mathcal{A}}$ that places m balls into m bins with *maximum load* $o(\log \log m)$ – this, of course, would contradict Vöcking's lower bound (Theorem 2 of [33]).

Let $\mathcal{A}$ denote the balls-and-bins strategy used in each time step of the time-step-isolated algorithm from the lemma statement. Let $\sigma = \langle x_1, x_2, \ldots, x_m \rangle$ be the sequence of m chunks (a.k.a. balls) that is accessed over and over again, and let $h_1(x_i), h_2(x_i), \ldots, h_d(x_i)$ denote the d servers (a.k.a. bins) where each chunk x_i is stored. For each chunk x_i, and $j \in [d]$, let

$$p_{i,j} = \Pr[\mathcal{A}(\sigma) \text{ routes } x_i \text{ to } h_j(x_i)].$$

Then we can construct a new online balls-and-bins strategy $\mathcal{A}'$ that works as follows on the same sequence $\sigma = \langle x_1, x_2, \ldots, x_m \rangle$ of ball arrivals: when choosing where to place ball x_i, it simply computes $p_{i,1}, p_{i,2}, \ldots, p_{i,d}$ (which depends only on $x_1, x_2, \ldots, x_i$), and places the ball in bin $h_k(x)$ where $k = \text{argmax}_j\, p_{i,j}$.

To bound the maximum load achieved by $\mathcal{A}'$, observe that $\mathcal{A}'$ only places a ball x_i in bin $h_k(x_i)$ if

$$p_{i,k} \geq 1/d.$$

Thus, if $\mathcal{A}'$ places s balls into some bin b, then the same bin b satisfies

$$\sum_{i \in [m]} \sum_{j \in [d]} p_{i,j} \cdot \mathbb{I}[h_j(x_i) = b] \geq s/d.$$

However, this implies that $\mathcal{A}'$ places, in expectation, at least s/d balls in bin b. We know from standard balls-and-bins lower bounds [33] that $\mathcal{A}'$ must (with high probability over σ) place at least $\Omega(\log \log m)$ balls into some bin. It follows that (with high probability over σ), there is some bin that, over time, receives an average of at least $\Omega(\log \log m)$ balls from $\mathcal{A}'$ per time step.

□

As an immediate corollary, we get the following:

Corollary 5.4. *It is impossible for a time-step-isolated strategy to achieve $d = O(1)$, $g = O(1)$, and a rejection rate of $O(1/m)$.*

6 RELATED WORK

The work in this paper is related to a diverse set of well-studied problems in the areas of load-balancing, queuing theory, balls-into-bins, and hashing.

Distributed key-value stores have been designed both for academia and commercial use [14, 16, 19, 20, 32, 36]. Most of this work is empirical and finds that handling online requests often does lead to load-balancing challenges [2, 13]. The most closely related work to ours is recent work by Wang et. al [34] (at PPoPP'23) which considers this problem, but in the case of $d = 1$ — that is, each chunk lives on at most one server and there is no replication. They show that when there is no replication and with a constant processing rate $g = O(1)$, no policy can achieve a rejection rate of $o(1)$. This leads them to consider a relaxation in which chunks can be moved over time from heavily loaded servers to lightly loaded ones and design a policy to achieve a small rejection rate in that manner.

Another model which considers load-balancing of requests that arrive online is the supermarket model [15, 24, 25, 31]. In this model, requests arrive at some stochastic rate (usually Poisson arrivals are assumed) and can be routed to any server. This research often considers the strategy where the incoming request looks at a small

number d of (random) servers and is routed to the server with the smallest queue among those — one can think of this as a queuing-theory version of the power of d choices. The stochastic arrivals of requests, and the fact that each request picks d random choices independently of past requests, means that the supermarket model cannot be used to address adversarial settings such as ours where the main technical challenge is reappearance dependencies.

Finally, balls into bins problems have been studied extensively with the power of d choices, starting from the classic paper by Azar et. al [3] which considers the task of assigning n balls to n bins, and then extended in many different directions by many different authors [1, 5, 10, 17, 21, 23, 26–28, 33, 35].

The most relevant line of balls-and-bins research to our context is work that deals with balls being inserted, deleted, and then (potentially) reinserted over time [5, 7, 11, 33]. In this setting, an oblivious adversary performs a sequence of ball insertions and deletions, where each ball x is associated with d random bins $h_1(x), h_2(x), \ldots, h_d(x) \in [m]$, and where the total number of balls in the system is never permitted to exceed some parameter k. The goal is to place balls in such a way that the maximum load across the bins stays small (close to k/m, ideally plus at most a polylogarithmic or doubly logarithmic term). This setting has yielded positive results in the cases where either $k = m$ [11] or $k = o(\log m)$ [7], and has led to a surprising negative result in the case of $k \gg m$ [5]. The negative result, in particular, which was shown at FOCS'23 by Bansal and Kuszmaul [5], says that in the heavily loaded case of $k \gg m$ (and specifically $m = O(1)$), almost all natural algorithms (specifically any so-called id-oblivious algorithm) fail to achieve good results in this setting – that is, there exists workloads that force the fullest bin to have $k^{\Omega(1)}$ more balls than the average load. Our results make use of quite different techniques from this line of work [5, 7, 11], and tackle reappearance dependencies that take a somewhat different form (notably, the same "ball" in our setting can be reinserted before it has even left the current queue), but nonetheless, all of these results can be viewed as part of a broader emerging theme: that reappearance dependencies can turn relatively simple stochastic problems into interesting and rich algorithmic problems which, in turn, often require the development of new algorithmic and analytical techniques.

ACKNOWLEDGEMENTS

This research was supported in part by the Harvard Rabin Postdoctoral Fellowship and the National Science Foundation with grants CCF-2106699, CCF-2107280, and PPoSS-2216971.

REFERENCES

[1] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. 1995. Parallel Randomized Load Balancing. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing* (Las Vegas, Nevada, USA) *(STOC '95)*. Association for Computing Machinery, New York, NY, USA, 238–247. https://doi.org/10.1145/225058.225131

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[3] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. 1999. Balanced Allocations. *SIAM J. Comput.* 29, 1 (1999), 180–200. https://doi.org/10.1137/S0097539795288490 arXiv:https://doi.org/10.1137/S0097539795288490

[4] Nikhil Bansal and Ohad N. Feldheim. 2022. The power of two choices in graphical allocation. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing* (Rome, Italy) *(STOC 2022)*. Association for Computing Machinery, New York, NY, USA, 52–63. https://doi.org/10.1145/3519935.3519995

[5] Nikhil Bansal and William Kuszmaul. 2022. Balanced Allocations: The Heavily Loaded Case with Deletions. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 801–812. https://doi.org/10.1109/FOCS54457.2022.00081

[6] LucaBecchetti Becchetti, Andrea Clementi, Emanuele Natale, Francesco Pasquale, and Gustavo Posta. 2015. Self-Stabilizing Repeated Balls-into-Bins. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. Association for Computing Machinery, New York, NY, USA, 332–339. https://doi.org/10.1145/2755573.2755584

[7] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Tiny pointers. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 477–508.

[8] Petra Berenbrink, Artur Czumaj, Matthias Englert, Tom Friedetzky, and Lars Nagel. 2012. Multiple-Choice Balanced Allocation in (Almost) Parallel. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, Anupam Gupta, Klaus Jansen, José Rolim, and Rocco Servedio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 411–422.

[9] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. 2000. Balanced allocations: The heavily loaded case. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 745–754.

[10] Hong Chen and Heng-Qing Ye. 2009. Asymptotic Optimality of Balanced Routing. *Operations Research* 60. https://doi.org/10.2307/41476346

[11] Richard Cole, Alan Frieze, Bruce Maggs, Michael Mitzenmacher, Andréa Richa, Ramesh Sitaraman, and Eli Upfal. 1998. On Balls and Bins with Deletions, Vol. 1518. 145–158. https://doi.org/10.1007/3-540-49543-6_12

[12] Richard Cole, Bruce M. Maggs, Friedhelm Meyer auf der Heide, Michael Mitzenmacher, Andréa W. Richa, Klaus Schröder, Ramesh K. Sitaraman, and Berthold Vöcking. 1998. Randomized Protocols for Low-Congestion Circuit Routing in Multistage Interconnection Networks. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 378–388. https://doi.org/10.1145/276698.276790

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[15] Felix Garcia-Carballeira and Alejandro Calderon. 2017. Reducing Randomization in the Power of Two Choices Load Balancing Algorithm. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*. 365–372. https://doi.org/10.1109/HPCS.2017.62

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

[17] Antonie S. Godtschalk and Florin Ciucu. 2012. Stochastic bounds for randomized load balancing. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (jan 2012), 74–76. https://doi.org/10.1145/2425248.2425267

[18] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39 (01 2009), 1543–1561. https://doi.org/10.1137/080728743

[19] Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, PPS Narayan, Adwait Tumbde, and Brian Cooper. 2012. The yahoo! cloud datastore load balancer. In *Proceedings of the fourth international workshop on Cloud data management*. 33–40.

[20] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[21] Dimitrios Los and Thomas Sauerwald. 2023. Balanced Allocations in Batches: The Tower of Two Choices. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 51–61. https://doi.org/10.1145/3558481.3591088

[22] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James Larus, and Albert Greenberg. 2011. Join-Idle-Queue: A Novel Load Balancing Algorithm for Dynamically Scalable Web Services. *Perform. Eval.* 68, 1056–1071. https://doi.org/10.1016/j.peva.2011.07.015

[23] M. Mitzenmacher. 1996. *The power of two choices in randomized load balancing*. PhD thesis. University of California, Berkeley.

[24] Michael Mitzenmacher. 1999. On the analysis of randomized load balancing schemes. *Theory of Computing Systems* 32, 3 (1999), 361–386.

[25] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

https://doi.org/10.1109/71.963420

[26] Debankur Mukherjee, Sem C. Borst, Johan S. H. van Leeuwaarden, and Philip A. Whiting. 2018. Universality of Power-of-d Load Balancing in Many-Server Systems. *Stochastic Systems* 8, 4 (2018), 265–292. https://doi.org/10.1287/stsy.2018.0016 arXiv:https://doi.org/10.1287/stsy.2018.0016

[27] Anis Nasir, Gianmarco Morales, Nicolas Kourtellis, and Marco Serafini. 2015. When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing.

[28] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*. 137–148. https://doi.org/10.1109/ICDE.2015.7113279

[29] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[30] Andréa Richa, Michael Mitzenmacher, and Ramesh Sitaraman. 2000. The Power of Two Random Choices: A Survey of Techniques and Results. https://doi.org/10.1007/978-1-4615-0013-1_9

[31] Andrea W Richa, M Mitzenmacher, and R Sitaraman. 2001. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization* 9 (2001), 255–304.

[32] Mehul Nalin Vora. 2011. Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 1. IEEE, 601–605.

[33] Berthold Vöcking. 2003. How asymmetry helps load balancing. *J. ACM* 50, 568–589. https://doi.org/10.1145/792538.792546

[34] Zhe Wang, Jinhao Zhao, Kunal Agrawal, He Liu, Meng Xu, and Jing Li. 2023. Provably Good Randomized Strategies for Data Placement in Distributed Key-Value Stores. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3572848.3577501

[35] Y. Xing, S. Zdonik, and J.-H. Hwang. 2005. Dynamic load distribution in the Borealis stream processor. In *21st International Conference on Data Engineering (ICDE'05)*. 791–802. https://doi.org/10.1109/ICDE.2005.53

[36] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*. 2653–2666.