Provably Good Randomized Strategies for Data Placement in Distributed Key-Value Stores

Zhe Wang
Washington University in St. Louis
zhe.wang@wustl.edu

Jinhao Zhao*
Washington University in St. Louis
jinhaoz@wustl.edu

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

He Liu
FoundationDB
heliu@apple.com

Meng Xu
FoundationDB
meng_xu@apple.com

Jing Li
New Jersey Institute of Technology
jingli@njit.edu

Abstract

Distributed storage systems are used widely in clouds, databases, and file systems. These systems store a large amount of data across multiple servers. When a request to access data comes in, it is routed to the appropriate server, queued, and eventually processed. If the server's queue is full, then requests may be rejected. Thus, one important challenge when designing the algorithm for allocating data to servers is the fact that the request pattern may be unbalanced, unpredictable, and may change over time. If some servers get a large fraction of the requests, they are overloaded, leading to many rejects. In this paper, we analyze this problem theoretically under adversarial assumptions. In particular, we assume that the request sequence is generated by an adversarial process to maximize the number of rejects and analyze the performance of various algorithmic strategies in terms of the fraction of the requests rejected. We show that no deterministic strategy can perform well. On the other hand, a simple randomized strategy guarantees that at most a constant fraction of requests are rejected in expectation. We also show that moving data to load balance is essential if we want to reject a very small fraction ($1/m$ where m is the number of servers) of requests. We design a strategy with randomization and data transfer to achieve this performance with speed augmentation. Finally, we conduct experiments and show that our algorithms perform well in practice.

Keywords: Load balance, Distributed key-value store

1 Introduction

Distributed key-value stores are used extensively on modern platforms, particularly in cloud applications [1, 8, 11, 17, 18], but also in large-scale databases [12, 20, 24, 36, 41] and distributed file systems [15, 22, 39]. In these applications, there is a large corpus of data items (e.g., key-value pairs, objects, files) stored across many servers. Clients of the system send requests, which access some particular data. Each request is routed to the server that has the items, and the server is responsible for processing this request and responding to the client. If the server is busy, the request may be queued.

For such systems, an important optimization criterion is ***throughput*** — the number of requests that the system can process on average per time period. If most requests end up accessing a small number of servers, most system capacity might be idle while a few servers' queues grow unboundedly. The client may care about ***latency*** — the amount of time they wait between sending a request and receiving a reply. Since long queues impact latency, distributed storage systems implement ***bounded*** queues for servers such that a server with a full queue rejects future requests.

The goal of the system is to ***accept*** or consume as many requests as possible, or inversely to ***reject*** as few requests as possible while keeping the queue size small — this is equivalent to maximizing throughput while keeping latency small. To avoid rejecting many requests (thereby reducing the system throughput), distributed storage systems try to balance load across servers. However, they have no direct control over the load, since requests are generated by clients and the request access pattern may change over time. Therefore, if all the "hot items" — data items that are being accessed frequently — are on the same server, that server will experience a high load. Avoiding overload may require load balancing via moving these items to other servers dynamically.

Load balancing algorithms for distributed storage systems generally have the following steps: (1) initially distribute data across servers in some manner; (2) each server has a queue with bounded size (decided by the algorithm) and rejects incoming requests if its queue is full; (3) do dynamic data distribution by choosing some data to move from an "overloaded" servers to other servers. The question considered in this work is how one should do these steps. This problem has been studied empirically using various heuristics based on past system behaviors or theoretically using queuing theoretic assumptions on request arrivals [9, 30].

In this paper, we consider this problem from a theoretical perspective. More formally, assume that the data is divided into n chunks that must be housed on m servers. At every time step, each server can process one request. We assume that the requests are generated by an adversarial process

*Corresponding author

PPoPP'23, Feb 25–Mar 01, 2023, Montreal, Canada
2023.

that generates m requests per time step and maximizes the number of rejected requests. Our goal is to understand what strategies with small queues may work against various adversarial assumptions. We have the following results:

Simple deterministic strategies work against random clients, but no deterministic strategy is good against an adversarial client: If the client is not adversarial and picks a random chunk to request in every time step, then any simple strategy that divides the chunks evenly across servers works well; however, if the client is adversarial, then, for any deterministic algorithm, there exists a request sequence such that the algorithm rejects most requests.

Simple randomized strategy works reasonably well against oblivious adversary: A simple randomized strategy that places chunks on servers uniformly at random and then never moves them performs does ok — that is, it accepts at least a constant fraction of the requests in expectation. In addition, this strategy provides a strong bound of rejecting only $O(1/m)$ fraction of the requests if either of the following is true: (1) if the adversary is weakened so that there is a bound on how frequently it can access the same chunk; or (2) servers have speed augmentation and each server consumes $O(\log m)$ requests per time step instead of 1.

Data transfer is useful: We show that no strategy without data transfer and constant speed can consume $1 - 1/m$ fraction of requests with an adversarial client. We also design an algorithm that starts with a randomized allocation and then moves chunks out of heavily loaded servers to balance the load. Given constant speedups of processing and data transfer, we show that this strategy consumes $1 - 1/m$ fraction of the requests even against a fully adversarial client.

Empirical evaluations are promising: To demonstrate the practicality of the system model considered in this work, we construct a case study using FoundationDB [41] to run realistic benchmarks on a real cluster. We conduct simulation experiments using adversarial and realistic workloads to evaluate the empirical performance of our proposed algorithms. Results show that randomized algorithms with and without data transfers only need a small speedup to achieve the same performance as an offline optimal algorithm.

2 System Model and Performance Metrics

In distributed data stores, we have m servers that store a large collection of items, such as key-value pairs, objects, or files. We can partition the data into n ($n > m^2$) data chunks, where a chunk is a collection of keys or a contiguous range of keys. The system distributes chunks to servers according to its load balancing policies. In this paper, we are going to abstract away the details of exactly how the data partition is done and assume that n and m are given and do not change over time. We also leave other problems such as data replication (and other database mechanisms, such as write-ahead log and multi-version control) as future work.

Clients send requests to the cluster online and we assume that each request accesses a single data chunk. When a request arrives, it is delivered to the server that hosts the appropriate data chunk. For analysis purposes, we divide the timeline into identical time slots, which have a length equal to the time to process a request. For simplicity, we do not distinguish the read, write, and read-range operations. Each server has a FIFO queue (first-in-first-out) with the maximum length of q to store the unserved client requests. When requests arrive, they are stored in the respective server's queue unless there is no space in which case the request is rejected. Since a server can consume one request in a slot, the cluster can consume at most m requests in the ideal case where the m requests access chunks at m different servers. Therefore, to make the workload feasible, we assume that at most m requests arrive at the cluster in a time slot and each of these requests accesses a different chunk.

During system initialization, the system allocates chunks to servers according to some load balancing algorithms, after which client requests are served. For the first few results in this paper, we assume that the allocation is fixed after initialization. However, some servers may be overloaded, and a load-balancing algorithm may want to move data from one server to another; therefore, later sections of the paper analyze algorithms that move data. We formally model the process of the data movement, named ***data transfer***, as follows. At any time, a server can be involved in one data transfer. In each transfer, at most s chunks to one other server, and this takes s time slots. In other words, transferring s or fewer chunks from one server to another takes s time; therefore, the transfer time is a stepped function, not linear. This models the fact that the latency of moving data from one server to another is often large, but the bandwidth is also large — therefore, sending or receiving 1 chunk vs. several chunks up to the bandwidth takes the same amount of time. On the other hand, moving two sets of chunks from or to two different servers takes $2s$ time slots, since the server must prepare the transfer and initiate the transfer. We use the single parameter s for modeling both the maximum number of chunks in a data transfer (with a total size below the bandwidth) and the latency of a data transfer. Our case study in Section 6 indicates that this model is reasonable on a real platform.

As noted in Section 1, the goal is to minimize the number of rejected requests, which is a measure of throughput.

Definition 1 (Throughput)**.** *Given a sequence σ of requests arriving over time, $A(\sigma)$ denotes the number of requests that are accepted by algorithm A on sequence σ.*

We assume the input sequence of client requests is generated by an oblivious adversary defined as follows.

Definition 2 (Oblivious adversary)**.** *The oblivious adversary knows how the online algorithm works, but it does not know the random choices made by the algorithm.*

For deterministic data allocation, the oblivious adversary always knows the exact locations of chunks at any point in time; therefore, unsurprisingly, we were able to show that no deterministic algorithm performs well. Most of this paper analyzes the performance of various randomized algorithms.

The theoretical performance of an algorithm can be analyzed by comparing its throughput with optimal throughput.

Definition 3 (Constant competitive)**.** *An online algorithm A is (constant) c-competitive, if $A(\sigma) \geq c \cdot OPT(\sigma)$ for any finite input sequence σ, where OPT is the offline optimal.*

An algorithm's performance is better when c gets closer to 1. While constant competitiveness is nice, we would prefer not to reject a constant fraction of the requests. Thus, we define the following, stronger performance criterion.

Definition 4 (Almost optimal)**.** *An online algorithm A is almost optimal, if $A(\sigma) \geq (1-O(1/m))OPT(\sigma)$ for any finite input sequence σ, where OPT is the optimal offline scheduler.*

For randomized algorithms, similar definitions apply except that we compare the expected throughput of the algorithm with the throughput of OPT. Note that in the best case for an offline optimal where all requests arriving in a time slot access different servers and are processed in this slot, all the $|\sigma|$ requests in a finite input sequence σ can be accepted by the optimal. Hence, an online algorithm is near optimal if it can accept $(1 - 1/m)|\sigma|$ requests for all input sequences.

Finally, some of our algorithms will require ***resource augmentation*** or ***speedup***. Resource augmentation implies that we allow the algorithm to perform certain operations faster than the optimal algorithm can. We consider two types of resource augmentation: (1) speed of processing requests on servers and (2) speed of data transfer. Often resource augmentation is necessary to achieve nontrivial results. However, the smaller the resource augmentation required, the better the algorithm. We generally want the resource augmentation factor to be no more than a constant.

3 Deterministic Policies

This section analyzes deterministic policies for allocating data chunks to servers. For warmup, we will first show that if the client requests access to random chunks, then any policy that evenly balances the number of chunks on each server performs well. On the other hand, no deterministic policy performs well against an oblivious adversary — for any deterministic policy, there exists an access pattern that causes the system to accept only a small number of requests.

Simple policies work well against random clients. We now do a simple analysis showing that all reasonable allocations work for random clients where each request picks a chunk to access uniformly at random and independently. Without loss of generality, we relax the assumption and allow multiple requests to access the same chunk even in the same time slot. For this random client, we consider a simple ***even allocation***, where n/m chunks are allocated to each server. The allocation is generated once and need not change. The following lemma is easy to prove using Chernoff bounds.

Lemma 1. *For a random client and an even allocation, in any $\log m$ consecutive time slots, the probability that more than $3 \log m$ requests arrive at a particular server is less than $1/m^2$.*

Lemma 1 leads to the following theorem.

Theorem 1. *Given a random client, an even allocation with speed 3 and queue size $q = 6 \log m$ accepts at least $(1-1/m)|\sigma|$ requests in expectation for any sequence of $|\sigma|$ requests.*

Proof. Divide the execution into rounds, where each round has $m \log m$ requests — therefore, each round has at least $\log m$ time steps, so a server can process $3 \log m$ requests with speed 3 from its queue. From Lemma 1, the probability that a server receives more than $3 \log m$ requests in a round is at most $1/m^2$. By union bound, in a particular round r, the probability that *at least one* server receives more than $3 \log m$ requests is at most $1/m$.

For a particular round r, we will prove via induction that the number of leftover requests from the previous round in any server's queue is at most $3 \log m$ for all servers at the start of every round. We have two cases. First, In round r, no server gets more than $3 \log m$ requests. In this case, the queue size of server a at the end of the round does not increase, since server a can process $3 \log m$ requests during a round if available. During this round, there can be at most $6 \log m$ requests in any queue and no requests are rejected. Second, in round r, some server get more than $3 \log m$ requests. We can pessimistically assume that all the $3 \log m$ requests that arrived in r are rejected maintaining the inductive hypothesis. Since requests are rejected only in the second type of rounds, which occur with a probability at most $1/m$, we reject at most $1/m$ fraction of requests in expectation. □

No deterministic policy works against an adversarial client. We now consider an oblivious adversary. Intuitively, since the algorithm is deterministic and the adversary knows the allocation, it can always overload some server. Here we argue that no deterministic policy, even if it moves data to load balance, can work well with small queues.

Theorem 2. *Consider a deterministic algorithm D running on a system with m servers, n chunks, max queue length q, data transfer time s, and constant speedup for both request processing and data transfer, where $n > m^2$. There exists a request sequence σ such that $OPT(\sigma)/D(\sigma) = \Omega(ms/(q+cs))$ where OPT is the offline optimal.*

Proof. Since $n > m^2$, at any time instant, there exists a server with more than m chunks. Since the algorithm is deterministic, the adversary always knows which server has more than m chunks and can send all m requests to these chunks.

Since the server can process only c chunks per time step, the queue will soon fill up causing most requests to be rejected.

Now, say that the D moves chunks. We divide the timeline into rounds. Each round has ms time slots, where s is the number of slots to perform a data transfer. Here is the adversary's strategy. At the start of round i, pick m chunks that are all in a particular server for D, say a, and send requests to these chunks only for the next s time steps. By the end of s steps, D can move s of these chunks away from a with q requests. With speed c for request processing, D can process at most cs requests in the s time steps and store at most q requests in the queue. Therefore, it accepts at most $cs + 2q$ of these ms requests. After this, for the rest of the round (the remaining $ms - s$ time steps), the adversary does not send any requests.

Since OPT knows the sequence, in the previous round $i-1$, it will set up to ensure that all these m chunks were on different servers so it can accept all requests. In the remaining $(m-1)s$ time steps of round i, OPT (in collaboration with the adversary) sets up for the next round. It knows which server will have at least m chunks at the start of the next round for D, so it distributes these m chunks across m servers for OPT so that OPT can answer all ms requests in the next round. The adversary can then repeat for the next round. □

4 Randomized Policies with No Transfers

In this section, we consider a simple randomized strategy that randomly allocates chunks to servers, which is similar to the game of throwing randomized balls into bins. We will show that it is constant competitive. We will also prove that no strategy that doesn't move data can be almost optimal against an oblivious adversary. In the next section, we will see a strategy that is almost optimal.

Definition 5 ($\mathbb{M}_{ball-bin}$)**.** *For n chunks and m servers, the algorithm picks a server for each chunk uniformly at random and independently. A $\mathbb{M}_{ball-bin}$ mapping denotes the mapping between chunks and servers.*

Upper Bounds for Balls into Bins. We will now prove a theorem against an adversarial client.

Theorem 3. *Given a system with m servers and n requests. For any sequence σ, say $E[B_\sigma]$ is the expected number of requests accepted by balls into bins allocation B. Then, $E[B_\sigma]/|\sigma| \geq 1 - 1/e$ even with queue size $q = 1$ and no resource augmentation.*

Proof. Consider the balls into bins allocation and consider a particular time step t when the client sends requests to $k \leq m$ distinct chunks. These k chunks were randomly thrown on m servers. Now consider a particular server a. The probability that exactly i of these k requests hit a is:

$$\binom{k}{i}\left(\frac{1}{m}\right)^i\left(\frac{m-1}{m}\right)^{k-i}$$

Therefore, the expected number of servers that get at least 1 request is at least $x = m - m\left(\frac{m-1}{m}\right)^k \geq (1-\frac{1}{e})k$. Any server that gets at least one request processes at least one request — therefore, at least x requests out of k total requests were consumed at this time step. Therefore, by adding over all time steps, we get the result. □

We see that the balls into bins allocation is constant competitive with no speed augmentation and with very small queues. What if we give it speed augmentation? We now show that with sufficient ($\Theta(\log m)$) speed augmentation, it is almost optimal.

Theorem 4. *Given a system with m servers and n requests. For any sequence σ, say $E[B_\sigma]$ is the expected number of requests accepted by the balls into bins allocation B. Then, $E[B_\sigma]/|\sigma| \geq 1 - O(1/m)$ with $\Theta(\log m)$ speedup on request processing and queue size $q = \Theta(\log m)$.*

Proof. Consider a particular time step t when the client sends requests to $k \leq m$ distinct chunks. These k chunks were randomly thrown on m servers when the allocation was done. Now consider a particular server a. The probability that at least x requests access this server at this time step is at most $\binom{k}{x}\left(\frac{1}{m}\right)^x \leq \frac{k^x}{x!}\left(\frac{1}{m}\right)^x = \frac{1}{x!} < 1/m^2$ for $x = c\log m$ with large enough c. At any time step, with $\Theta(\log m)$ speed, $x = \Theta(\log m)$ requests can be processed by a server. Therefore, the balls into bins allocation can process all the k requests that arrive on that time step with a probability at least $1-1/m$. Even assuming pessimistically that all requests at other times are rejected, summing over time still gives us the result. □

Lower bound on algorithms with no data transfer. We saw that the balls into bins algorithm is almost optimal with $\Theta(\log m)$ speed; however, what about constant speed? We now show a lower bound, saying that any algorithm without moving data cannot be almost optimal with constant speed.

Theorem 5. *Given any policy D for allocating chunks, for any queue with length q, a constant speedup of processing, if $n > m^2$, then there exists an input sequence, such that $E[D]/E[Opt] \leq p$ where $p < 1$ is a constant.*

Here is the adversary's strategy: It simply selects m different chunks at random and repeatedly requests these chunks in every time slot. The challenge is to show that there is ***no*** randomized strategy that can provide an almost optimal acceptance ratio in expectation for this sequence. To do this, we will define a concept of a ***group***, which will allow us to extract a common property of all distributions.

Definition 6 (group)**.** *Let a T-*group *consists of* exactly *T chunks that are in the same server, and different groups do not have any chunks in common.*

We want all groups to have the same number of chunks in order to compute probabilities. If a server has at least T chunks, we create groups with T chunks each and leave the

leftover chunks ***ungrouped***. Repeatedly put T ungrouped chunks into groups until we have fewer than T ungrouped chunks. The following observation just says that there are a large number of groups.

Observation 1. *For any position of n chunks in m servers, the number of $(n/2m)$-groups is at least m.*

We now only consider grouped chunks and we will show that a large number of requests sent to grouped chunks will be rejected. In particular, we will first show that the probability that a group gets a large number of requests is not too small.

Lemma 2. *Given a specific distribution of the chunks, and suppose the adversary randomly selects m chunks. Then for $k = o(m)$ and $n > m^2$, the probability that an $(n/2m)$-group has at least k requests is greater than $p = \frac{1}{8^k k!} \cdot e^{-\frac{3}{2}}$. We can say that these groups are **overloaded**.*

Proof. Since the adversary randomly chooses m distinct chunks, the number of chunks in each group follows a *Hypergeometric Distribution*. In particular, consider $k = o(m)$ and group size $T = n/2m$. We can assume that $m-k > m/2$ and $T-k > T/2$ and $m-k < n/m$. We can then bound the probability that a particular group has k chunks among the m chunks by the following:

$$\begin{aligned}
\frac{\binom{T}{k}\binom{n-T}{m-k}}{\binom{n}{m}} &= \frac{\frac{T!}{k!(T-k)!} \cdot \frac{(n-T)!}{(m-k)!(n-T-(m-k))!}}{\frac{n!}{m!(n-m)!}} \\
&= \frac{1}{k!} \cdot \frac{\frac{T!}{(T-k)!} \cdot \frac{(n-T)!}{(n-T-(m-k))!} \cdot \frac{m!}{(m-k)!}}{\frac{n!}{(n-m)!}} \\
&> \frac{1}{k!} \cdot \frac{(T-k)^k \cdot (n-T-(m-k))^{m-k} \cdot (m-k)^k}{n^m} \\
&> \frac{1}{k!} \cdot \frac{(\frac{m}{2})^k \cdot (\frac{T}{2})^k \cdot (n(1-\frac{1}{2m}-\frac{1}{m}))^{m-k}}{n^m} \\
&= \frac{1}{8^k k!} \cdot (1-\frac{3}{2}\cdot\frac{1}{m})^{m-k} > \frac{1}{8^k k!} \cdot e^{-\frac{3}{2}},
\end{aligned}$$

This proves the lemma. □

Now we are able to give a proof for Theorem 5. Since the adversary repeatedly requests the same m chunks, overloaded groups remain overloaded. Therefore, regardless of the queue size, servers with overloaded groups will eventually reject many of their requests.

Proof. According to lemma 2, each $(n/2m)$-group is overloaded with probability p — they get at least k requests on every time step. Therefore, with any speedup smaller than k, these groups remain overloaded forever since their total capacity is smaller than their average load. Therefore, these chunks will reject at least one request per time step. Since the expected number of overloaded groups is $mp = \frac{m}{8^k k!} \cdot e^{-\frac{3}{2}}$, with speed $< k$, at least mp requests are rejected on every time step within expectation while the optimal algorithm can handle all m requests. Therefore, the fraction of the requests that any algorithm accepts is p, and p is a constant for any constant k. □

Constraining the request sequence. The argument in the previous section indicates that the worst-case workload for any randomized strategy that does not move data is an adversary which requests the same m chucks repeatedly. However, this is often not what happens in the real world. In this section, we show that we can do better if the adversary can not request the same chunks repeatedly — we assume that the client issues at most one request to a particular chunk in any consecutive $\log m$ time steps and show that balls into bins strategy is almost optimal against this adversary.

We divide time into phases of size $\log m$. The following Lemma is similar to Lemma 1, except that the randomness comes from the randomized allocation rather than the client.

Lemma 3. *Against a constraint adversary which can not send more than one request to the same server in one phase, the probability that any server gets more than $3\log m$ requests in a phase is at most $1/m^2$.*

With Lemma 3, we can prove Theorem 6 in a manner very similar to the proof of Theorem 1.

Theorem 6. *Given a queue with size $q = O(\log m)$ and a constrained adversarial input σ, the expected number of requests consumed by balls into bins policy is $E[B] \geq (1-1/m)\sigma$.*

The result indicates that a workload is great if requests are not successively and repeatedly access to the same chunk.

5 Randomized Policies with Transfers

In this section, our goal is to design a policy that is almost optimal (consumes $1-O(1/m)$ fraction of the requests) with constant speedup and without placing any restrictions on the input sequence. In particular, we propose an algorithm, say Y which satisfies the following strong guarantee.

Theorem 7. *Say we have a constant speedup of data movement and a constant speedup of processing on the system Y with the queue length $q = \Theta(s\log^2 m)$, where s is the number of time slots to complete a data transfer. For any input sequence σ, the expected number of requests consumed by Y is at least $(1-O(1/m^2))\sigma$.*

An overview of the system Y: In prior sections, we have seen that while deterministic policies can not be shown to be constant competitive (even allowing data movement) (Theorem 2), simple randomized policies such as allocating chunks uniformly at random can achieve constant competitiveness (Theorem 3). In particular, with balls into bins randomization, a constant fraction of the requests can be consumed on each time step. However, since some servers get more than a constant number of requests (some servers get $\Omega(\log m)$ requests) on each time step, these servers are overloaded and

the adversary can keep them overloaded — therefore, with at most constant speed and with no restrictions on the input sequence, no randomized policy without data movement can be almost optimal (Theorem 5).

To address this issue, we will design a system, we call it, Y which moves data to get a good load balance. In particular, we start with a randomized balls into bins allocation just like the previous sections. Therefore, all chunks have a ***home server*** where they were allocated through balls into bins. However, when the queue of the server contains a request that has been in the system for $\Theta(s \log m)$ time, this indicates that this server is overloaded and can not handle all the requests being sent to it. At this point, a ***batch movement process*** is triggered on this server and the chunks in the server that have pending requests are distributed to other servers so that these old pending requests can be handled efficiently by other servers. Once these pending requests are handled, these chunks are moved back to their home server so that the original randomized allocation is restored.

Modeling assumptions: For simplicity in analysis, we will make some modeling assumptions that do not impact the overall result. In particular, we will assume that each server has two processors: a ***primary processor*** which consumes the requests that arrive at this server from the client and a ***secondary processor*** which consumes the requests that are sent to it from other servers for load balancing purposes from other processors. Each server has its own queues, where the primary and secondary queues are both of size $\Theta(s \log^2 m)$. Note that this does not impact the theorem statement since we allow for constant speedup — if we allow for processor speed of ρ^p, then each of the processors can run at half the speed. Similarly, the queue length can be split among the two queues while only impacting constant factors. In this analysis, we will not try to optimize constant factors; therefore, the constant factors computed will be large. In the evaluation section, we will see that the algorithm requires a quite small constant in practice to perform very well.

To restate the algorithm using these terms: when a request arrives at a server from a client, it is put in the primary queue. If the primary queue is full, then the request is rejected. If the age of the oldest request in the primary queue is $6s \log m$, the server triggers a batch movement process, and any chunk that has a request in the primary queue is moved to other servers and their corresponding requests are moved to those servers' secondary queues (***move out phase***). These moved requests are processed by the target processor's secondary server. Once a server's secondary queue is empty, the chunks that have been moved there are sent back to their home servers (***move back phase***). We will define the precise policy of movement momentarily, but let us first consider what the challenges are in designing this policy.

Challenges: Intuitively, the system Y should work since (1) it moves the requests out of the primary queue in a timely manner to avoid filling the primary queue; (2) it spreads the requests among the secondary queues, which efficiently consumes the requests and avoids filling secondary queue. However, there are a few challenges. First, a batch movement process might take a long time if the batches are large. When a server triggers a batch movement, many chunks in that server may have pending requests in the primary queue and all these chunks must be moved. Recall, from Section 2, that it takes s time to move at most s chunks between two particular servers — we call this one operation a ***transfer***. Therefore, depending on how many transfers are required, a full batch movement may take time. This can cause the queue of the server to get longer while the batch process is executing as well as blocking other batch processes from starting. Second, we may get unlucky and a server may get a very large number of requests from the client at the same time step. When this happens, the primary queue can become overloaded very quickly, potentially causing downstream effects. Third, the batch process is deterministic — therefore, in principle, the adversary can guess the location of chunks when they are being processed away from their home server and can overload the servers where these chunks are located by sending too many requests there.

Algorithm Description. We can now describe Y and how it handles these challenges. The queue size of each server is $\Theta(s \log^2 m)$ for a sufficiently large constant hidden in the Θ-notation.

(I) Batch trigger: A batch process at a particular server a is triggered when the oldest request at this server that is not already a part of an older batch process is $6s \log m$ old.

(II) Batch start: In Y, batch processes may not start as soon as they are triggered; they are executed in order. Say, the batch process i is triggered before the batch process j is triggered. If the processes are triggered on the same server, then j executes after i completes since the same chunks may be involved in both. If they are on different servers, they can execute in parallel. Our distribution algorithm ensures that a server is involved in at most transfer at a time.

(III) Data distribution during moveout: Once a batch is triggered, we know which chunks are part of the batch — that is which chunks have pending requests which are part of this batch. These chunks are divided into $\log m$ packages for transfer so that each bucket has $O(s)$ chunks and at most $O(s \log m)$ requests. We later show that this is possible with high probability. Therefore, each of these packages can be moved to destination servers using one transfer each. When the batch starts, these buckets are moved to $\log m$ different servers and the corresponding requests are moved to those server's secondary queues.

(IV) Process the chunks and move back: The secondary processor of the target server processes the requests from the secondary queue in order. Once all requests of a particular

transfer have been processed, the chunks are moved back to their home server.

(V) Request handling during batch: Even when a chunk has been moved to a different server for processing its pending requests, any new requests to this chunk are still sent to its home server's primary queue and these are processed once the chunk has moved back to its home server.

(VI) Flow control and batch cut-off: In order to avoid some boundary conditions, we perform some controls: (1) If a single server receives more than $2\log m$ requests in a single time step, then some of the requests are immediately rejected to ensure that only $2\log m$ requests are added to any primary queue on any single time step. (2) If the server has more than $24s\log m$ chunks with requests in the primary queue, the server only moves $24s\log m$ chunks with the most requests, and the server rejects the requests of unmoved chunks.

Causes of rejection. We want to show that this process rejects at most $O(1/m^2)$ fraction of the requests in expectation. Note that requests can be rejected due to the following reasons: (1) flow control and batch cut; (2) rejection from the primary queue if it becomes full; and (3) rejection from the secondary queue if it becomes full. We will show that the first two reasons cause few rejects in expectation and the last reason causes no rejections for a good load balancing policy. However, in order to show this, we must first bound the time it needs to complete a batch process.

Bounding the execution time of a batch process. We first bound the execution time of a batch process — the time between the triggering and completion of a batch process. We will divide time into phases of size $3s\log m$. We say that X_i is the set of batch processes that were triggered during phase i. We will prove the following key Lemma.

Lemma 4. *Any batch process that was triggered during phase i will be completed by the end of phase $i+1$ with constant speedup on both processing speed and transfer speed.*

In order to prove this lemma, we first prove some supporting lemmas. First, we recall that a batch process is triggered when the oldest request in the queue which is not already part of the batch process has been in the queue for $6s\log m$ time. Therefore, a batch process can contain requests from at most $6s\log m$ time steps. We can bound the number of requests that are part of a batch process as follows:

Lemma 5. *Given the balls into bins allocation, constant $c \geq 1$ and $s < m/(c\log m)$. In $6s\log m$ consecutive time slots, the probability that a particular server receives $12s\log^2 m$ or more requests is less than $1/m^2$.*

Proof. Note that the client can send requests to the same chunk in different time steps, but in one time slot it must send m different requests. In each time step, for a particular server a, say X_a^i is the random variable representing the number of requests that hit this server at time step i. For $x = 2\log m$, we have

$$\Pr(X_a^i \geq x) \leq \binom{m}{x}\left(\frac{1}{m}\right)^x \leq \frac{m^x}{x!}\left(\frac{1}{m}\right)^x = \frac{1}{x!} < \frac{1}{m^3}$$

The last equation works when $m > 4$. Therefore, we can sum up X_a^i through all $cs\log m$ steps and use Bernoulli's inequality to get the bound we need. □

We can also bound the number of chunks involved in a batch process by a method very similar to Lemma 1.

Lemma 6. *Given a balls into bins allocation, the total number of chunks from a particular server that can be requested within $6s\log m$ time is $24s\log m$ with probability of at least $1-1/m^2$. Therefore, the total number of chunks involved in a batch process is $24s\log m$ with probability of $1-1/m^2$. Hence, each batch process causes $\log m$ transfers to different targets.*

Given these lemmas, we can define a ***transfer packaging policy***. Recall that each data transfer can transfer up to s chunks to a particular server. Once we have defined a batch with at most $O(s\log^2 m)$ requests and $O(s\log m)$ chunks, we package these into $\log m$ transfers greedily. We keep adding chunks to a transfer until either the number of requests in the transfer is greater than $c_1 s\log m$ for some constant c_1 or the number of chunks in the transfer is greater than $c_2 s$. At this point, this transfer is complete and we start a new transfer. Due to the previous two lemmas, the total number of transfers for a particular batch process is at most $\log m$ for suitable choices of c_1 and c_2. Therefore, we have the following corollary:

Corollary 7.1. *The amount of time it takes to move all the chunks that are part of the same batch process to $\log m$ target servers is $s\log m$ assuming that we have data movement speed-up of 24 — that, we can move $24s$ chunks in s time.*

We can also bound the number of requests in the secondary queue under certain conditions. Consider batch processes in $X_i \cup X_{i+1}$ — the batch processes triggered during two consecutive phases. We want to bound the number of requests that end up in the same server's secondary queue due to these batch processes.

Lemma 7. *Considering only requests which are part of batches from $X_i \cup X_{i+1}$, the number of these requests that end up in a particular secondary server's queue is at most $c_3 s\log m$ for some constant c_3.*

Proof. All the requests that are part of batch processes in $X_i \cup X_{i+1}$ arrive within a time interval of $12s\log m$ since any request which is in batch from X_i must have arrived at most $6s\log m$ time before the beginning of X_i. Therefore, these batches can contain a maximum of $12sm\log m$ requests. Since requests are evenly balanced across servers during the batch movements and we have m servers, no server has more than $c_3 s\log m$ requests for a suitably large c_3. □

We now observe the following invariant about consecutive phases since a batch is triggered when the oldest request which is not part of a batch arrived $6s \log m$ time ago.

Observation 2. *Consider any two consecutive phases i and $i+1$, and consider the set $X_i \cup X_{i+1}$ — the set of batch processes triggered during these phases. No two batch processes in this set can be triggered by the same server.*

Now we can prove Lemma 4 via induction.

Proof. The base case is trivial since the first phase doesn't trigger any batch processes. For Inductive Hypothesis, assume that all batch processes triggered during phase i completed by the end of phase $i+1$. Therefore, all batch processes triggered during phase $i+1$ are ready to start at the beginning of phase $i+2$ since all prior batch processes at their respective servers have completed.

None of the batch processes triggered during phase $i+1$ have the same source (Observation 2). In addition, they trigger $\log m$ transfers each (Corollary 6). Therefore, there are a total of at most m sources and at most $m \log m$ targets of the transfers — these can be scheduled in $s \log m$ time without source or destination conflicts with sufficient large constant speedup in data transfer speed.

Now consider the processing of these requests by the secondary server. The secondary server can only contain requests due to batch processes triggered during phase $i+1$ and phase $i+2$ (since phase $i+2$ has started, some of the batch processes in this phase may have also started). By Lemma 7, the total number of requests in any secondary queue due to requests from these two phases is at most $c_3 s \log m$ for some constant c_3. Therefore, again, with sufficient constant speedup on processing, all these requests can be processed by the secondary server in the next $s \log m$ steps. Finally, all the chunks from batch processes in phase $i+1$ must move back and again, which can be done in $s \log m$ time by using a reverse schedule from the move-out schedule. Therefore, in at most $3s \log m$ time after the start of phase $i+1$ — that is, at the end of phase $i+2$ — all these batches have completed. □

Bounding the number of rejected requests. We can now show that the number of rejected requests is small. Below is the proof of Theorem 7.

Proof. Recall that requests can be rejected due to the two control strategies. First, Lemma 6 implies that the probability that a batch has more than $24s \log m$ chunks involved is less than $1/m^2$. Since rejecting a request due to the second policy happens only on these batches, this probability is also less than $1/m^2$. Second, consider the case where $2 \log m$ requests arrive at the same server. For a particular server a, say X_a is the random variable representing the number of requests that hit this server at this time step. Denote $x = 2 \log m$, we have

$$\Pr(X \geq x) \leq \binom{k}{x} \left(\frac{1}{m}\right)^x \leq \frac{k^x}{x!} \left(\frac{1}{m}\right)^x = \frac{1}{x!}$$

For $m > 4$ we have $\frac{1}{x!} < 1/m^3$. According to the union bound through all the servers, the probability that it is rejected due to the first policy is less than $1/m^2$.

In addition to the control strategies, a request may be rejected if the primary queue is full. However, recall that each batch process completes in time at most $O(s \log m)$ and due to Lemma 5 the total number of requests that arrive in this time period is at most $O(s \log^2 m)$ with probability $1 - 1/m^2$. Since the primary queue has this capacity, the probability of a request being rejected due to this is small. Finally, Lemma 7 indicates that requests can never be rejected from the secondary queue since the number of items in it are at most $O(s \log m)$. □

6 Empirical Evaluations

In this section, we evaluate the practicality of our theoretical study on load balancing in distributed key-value stores via a case study real-world database cluster and extensive simulations based on statistics measured from realistic workloads.

6.1 A Case Study

Our case study platform is an Amazon Kubernetes Service cluster running FoundationDB [41], which is an open-source distributed database supporting transactional key-value stores. We develop experimental components in FoundationDB to measure the runtime statistics and customize workloads to mimic real-world applications. The case study aims to verify the applicability of the theoretical model for the load balancing problem in distributed key-value stores and collect statistics to be used in simulation experiments.

Experimental design. We set up a FoundationDB cluster on the Amazon Kubernetes Service with 70 storage servers and one data distributor, managed by the FoundationDB Operator. Each storage server runs in a dedicated Kubernetes pod with Amazon EBS storage and contains more than 100 data chunks. The RocksDB [5] is used as the storage engine.

The storage layer of the FoundationDB implementation that supports snapshots, including RocksDB, works as follows. The keys (values) are persistent in RocksDB checkpoints. Each checkpoint, denoted as a data chunk in previous sections, is stored as a physical file on the disk. Each key is persistent in one checkpoint file. A data transfer from server i to server j is implemented as fetching and transmitting at most s checkpoint files from server i to j.

In FoundationDB, a client uses transactions to interact with the storage. A transaction contains a series of requests (operations) on keys, such as read, write, and scan (read range) requests. Since this work focuses on the storage layer, we regard transactions as the proxy of clients.

Our case study runs the realistic workload using the Yahoo! Cloud Serving Benchmark (YCSB) [10], with an extension that client requests have a 90/10 Read/Write ratio, which mimics the pattern in real-world applications. We run the workload for one hour containing more than one million requests, from which we randomly sample and measure 100K requests with specific properties under consideration.

Since the data distributor issues a data transfer and monitors it until the data chunks complete moving, we measure the data transfer latency inside the data distributor. Hence, the latency measurement is accurate without needing clock synchronization between storage servers. When recording a data transfer, we collect the number of bytes in the moved data chunks, its end-to-end latency, the source storage server, and the destination storage server.

Data transfer latency. Recall that our theoretical model assumes that the latency of a data transfer does not increase with the number of moved chunks, as long as the total size is below the network bandwidth. To validate this assumption, we use our FoundationDB cluster to measure the data transfer latencies under different settings: (1) moving chunks with different total data sizes from one server to another server, (2) receiving chunks in multiple data transfers from different servers at the same destination server, and (3) sending chunks in multiple data transfers from the same source server to different servers.

Figure 1(a) presents the latency of standalone data transfers with different total sizes. We can see that the latency does not increase with larger data sizes. This is mainly because the network bandwidth between storage servers is approximately above 1GB per second, which is larger than the total size. Thus, the latency is dominated by the CPU overhead of the storage server to prepare for the data transfer, which is independent of the chunk size.

For multiple data transfers to the same destination server or from the same source server in Figures 1(b) and 1(c), we can see that the latency increases roughly linearly with an increasing number of data transfers. This is because the storage server can only process the data transfers sequentially, which is consistent with our theoretical model.

Data transfer latency vs. request processing latency. For analyzing theoretical bounds, the time to process a client request is considered as 1 time slot, while the time to perform data transfer is denoted as s time slots. Our empirical measurements show that the request processing latency and data transfer latency are at the magnitude of 0.001 and 0.1 seconds, respectively. Besides, the read and scan requests have smaller latencies than that of write. Hence, the ratio s is around 100 to 1000 in practice. Thus, our simulation experiments below use $s = 100$.

6.2 Simulation Experiments

We conduct simulation experiments with different workloads to evaluate our proposed data distribution policies.

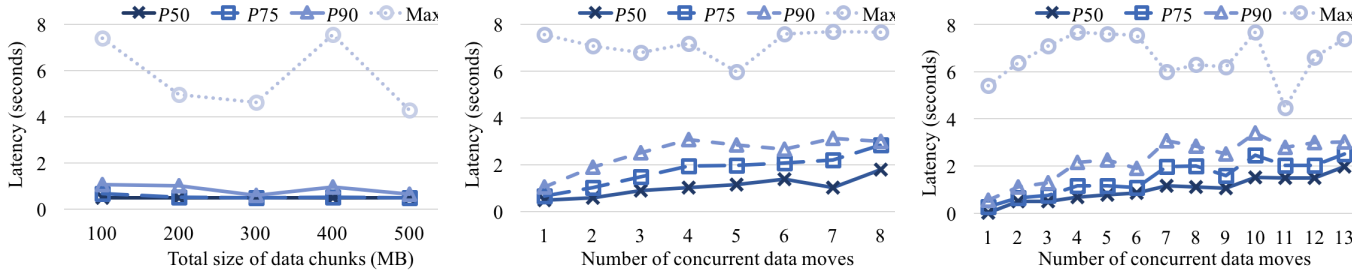
Experimental design. We implement a simulator for a FoundationDB cluster with m storage servers, where n data chunks are distributed among the servers according to the chosen policy. During a simulation, the simulator generates requests that access some servers following some workload pattern. When adding a generated request to the server's queue, the simulator rejects (drops) the request if the number of requests waiting in the queue is already equal to the queue length q. Note that the policy in Section 5 utilizes two queues. In this case, a request is rejected if both queues with a total size q are full.

We experimented with two request workloads: Adversary and Zipfian. The adversarial workload always generates requests that access the same set of m chunks during the simulation, which is especially difficult for policies without knowing this access pattern. In comparison, the Zipfian workload generates requests that access chunks following the Zipfian distribution [30] with the parameter $a = 2$, which represents more realistic workloads. For each setting, we run 10 simulations, each with 4.5 million generated requests, measure the request rejection ratios and report the median.

Policies and baselines. We implemented our proposed randomized policy without data transfers (labeled Random) and with data transfers (labeled DataMove). To examine how speedup of request processing and data transfers improves the performance of our policies given the same workload, we also enable the simulator to have different speeds. In particular, at speed 1, a server processes one request in a time slot. At speed 2, for instance, a server processes two requests in a time slot. In the simulation, there is no speedup of data transfer.

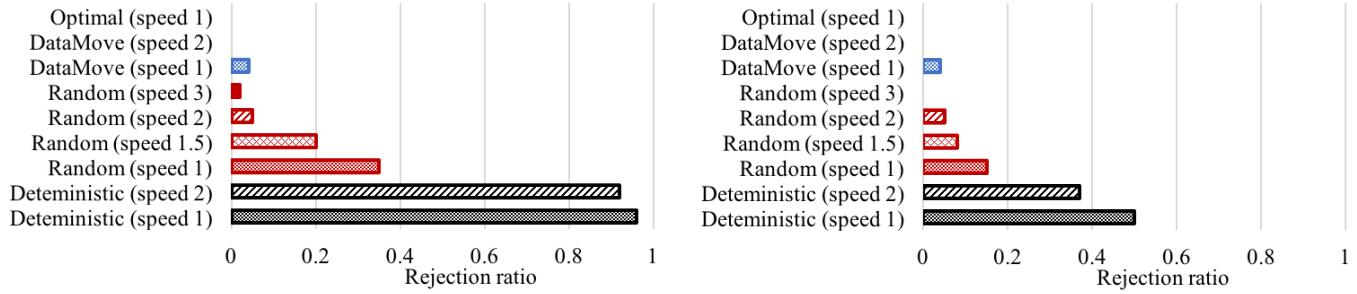
As a baseline comparison, we implemented and evaluated a policy that deterministically distributes data chunks to servers. Additionally, if requests evenly access all servers, the requests generated at the beginning of a time slot can all be fully processed in this time slot at speed 1. Therefore, the lower bound of an optimal policy can achieve zero rejection.

Evaluation results. Figure 2 shows the rejection ratios under the adversarial and ZipFian workloads, where the queue lengths are set to $q = 200s \log m$ and $q = 20s \log m$, respectively. The queue length is chosen so that we can clearly observe the performance differences under different policies. Simulation results show that randomized policy significantly outperforms deterministic policy. Adding data transfers to the randomized policy further reduces the rejection ratios for both workloads. Moreover, we can see that our proposed Random and DataMove policies can already achieve zero rejection with a speedup of 2 and 3, respectively. This result reveals that the required speedup of our policies in practice



(a) One standalone data transfer with increasing total data sizes (b) Increasing numbers of data transfers from multiple servers to the same destination (c) Increasing numbers of data transfers from the same source to multiple servers

Figure 1. 50%, 75%, 90%, and maximum latencies of data transfer under different settings.



(a) Adversarial workload (with queue size $q = 49069$) (b) ZipFian workload (with $q = 4907$ & ZipFian parameter $a = 2$)

Figure 2. Rejection ratio of deterministic policy (Deterministic), randomized policy (Random), and randomized policy with data transfers (DataMove) given different speeds for different workloads. System setting: $m = 30$, $n = 1000$, $s = 100$.

can be significantly smaller than that in the theoretical results. We have also run experiments with larger s, m, and n and observed similar performance trends.

7 Related work

Both the problem and the solution are related to many topics studied by researchers. Here we provide a brief overview of some of the related work.

Distrbuted key-value stores have been designed both for academia and commercial use [12, 15, 23, 24, 36, 41]. Most of these systems are evaluated empirically and most of this work finds that handling online requests that might be skewed towards certain keys is often an important challenge [2, 10]. Most of the theoretical analysis of these systems is done by making stochastic assumptions on the arrival pattern of requests [28, 29, 32]. In this paper, we take a different tack and analyze these systems under adversarial inputs.

Another related area of research is distributed hash tables, which have received extensive theoretical and empirical investigation. For instance, consistent hashing [21, 27] is widely used to partition the data in distributed hash maps [12, 24, 34]. Randomization is widely used to build and maintain the distributed hash tables [13, 25, 38]. In addition, some distributed hash tables consider network topology in the design of the hash table itself [3, 33]. The problem we consider in this paper is also related to general load balancing problems where tasks can be dispatched to various servers. The techniques used in this paper such as balls-into-bins analysis were originally developed in the load balancing context. These games have been extensively analyzed for various metrics [4, 16, 31]. In addition, variations of these games such as power of two choices [14, 26, 29] have also been analyzed extensively and sometimes used in the design of distributed hash tables. The problem we considered in this paper is different but related, to the distributed hash table problem as well as the load balancing problem. In particular, the model for queuing and the model for moving data from one location to another is generally different in distributed hash tables which leads to different algorithmic and analytical challenges.

Since most online systems are dynamic where the system load changes and the keys which are accessed frequently change over time, data migration is an important tool in this system design. There has been extensive empirical work for designing data migration schemes and evaluating them in a diverse range of systems [6, 7, 19, 23, 35–37, 40, 41] for various distributed workloads in key-value stores and distributed databases. Most of this work is empirical or uses stochastic assumptions on data arrivals for analysis.

8 Conclusion

We have explored the problem of load balancing in distributed storage systems, assuming an oblivious adversary

sends requests. With reasonable queue sizes and good algorithm designs, it is possible to consume almost all requests in expectation. Several open problems remain. First, we assume that each chunk is stored in at most one server. Real systems often store each chunk on multiple servers via replication to aid the load balancing and recovery from faults. We will design algorithms to support replication. Second, we assume an oblivious adversary that gets no information from the system once it runs. One can imagine an adaptive adversary that can glean the requests' distribution from the rejection pattern or reply latencies. Finally, we would like to explore if there are algorithms that can provide results similar to ours but using smaller queue sizes and thus smaller latencies.

Acknowledgments

This research was supported, in part, by the National Science Foundation (USA) under Grant Numbers CNS–1948457, CCF–1733873 and CCF–1725647. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[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] John Augustine, Soumyottam Chatterjee, and Gopal Pandurangan. 2022. A Fully-Distributed Scalable Peer-to-Peer Protocol for Byzantine-Resilient Distributed Hash Tables. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 87–98.

[4] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. 2008. On weighted balls-into-bins games. *Theoretical Computer Science* 409, 3 (2008), 511–520.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[6] Rajat Chaudhary, Gagangeet Singh Aujla, Neeraj Kumar, and Joel JPC Rodrigues. 2018. Optimized big data management across multi-cloud data centers: Software-defined-network-based analysis. *IEEE Communications Magazine* 56, 2 (2018), 118–126.

[7] Yue Cheng, Aayush Gupta, and Ali R Butt. 2015. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[8] Christos Chrysafis, Ben Collins, Scott Dugas, Jay Dunkelberger, Moussa Ehsan, Scott Gray, Alec Grieser, Ori Herrnstadt, Kfir Lev-Ari, Tao Lin, et al. 2019. Foundationdb record layer: A multi-tenant structured datastore. In *Proceedings of the 2019 International Conference on Management of Data*. 1787–1802.

[9] Prem C Consul and Gaurav C Jain. 1973. A generalization of the Poisson distribution. *Technometrics* 15, 4 (1973), 791–799.

[10] 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.

[11] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[12] 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

[13] Osman Durmaz and Hasan Sakir Bilge. 2019. Fast image similarity search by distributed locality sensitive hashing. *Pattern Recognition Letters* 128 (2019), 361–369.

[14] Felix Garcia-Carballeira, Alejandro Calderon, and Jesus Carretero. 2021. Enhancing the power of two choices load balancing algorithm using round robin policy. *Cluster Computing* 24, 2 (2021), 611–624.

[15] 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.

[16] P Brighten Godfrey. 2008. Balls and bins with structure: balanced allocations on hypergraphs. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Citeseer, 511–517.

[17] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[18] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An intelligent data lake system. In *Proceedings of the 2016 international conference on management of data*. 2097–2100.

[19] DJ Hemanth et al. 2020. Efficient Data Migration Method in Distributed Systems Environment. (2020).

[20] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[22] A Kala Karun and K Chitharanjan. 2013. A review on hadoop—HDFS infrastructure extensions. In *2013 IEEE conference on information & communication technologies*. IEEE, 132–137.

[23] 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.

[24] 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

[25] Cong Leng, Jiaxiang Wu, Jian Cheng, Xi Zhang, and Hanqing Lu. 2015. Hashing for distributed data. In *International Conference on Machine Learning*. PMLR, 1642–1650.

[26] Malwina J Luczak and Colin McDiarmid. 2005. On the power of two choices: balls and bins in continuous time. *The Annals of Applied Probability* 15, 3 (2005), 1733–1764.

[27] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. 2018. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 587–604.

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

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

[30] David MW Powers. 1998. Applications and explanations of Zipf's law. In *New methods in language processing and computational natural language learning*.

[31] Martin Raab and Angelika Steger. 1998. "Balls into bins"—A simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer, 159–170.

[32] 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.

[33] Renisha P Salim and R Rajesh. 2022. A Framework for Integrating the Distributed Hash Table (DHT) with an Enhanced Bloom's Filter in MANET. *International Journal of Advanced Computer Science and Applications* 13, 2 (2022).

[34] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking* 11, 1 (2003), 17–32. https://doi.org/10.1109/TNET.2002.808407

[35] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[36] 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.

[37] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. 2020. {MAPX}: Controlled Data Migration in the Expansion of Decentralized {Object-Based} Storage Systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 1–11.

[38] Shengnan Wang, Chunguang Li, and Hui-Liang Shen. 2018. Distributed graph hashing. *IEEE transactions on cybernetics* 49, 5 (2018), 1896–1908.

[39] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 307–320.

[40] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 31–31.

[41] 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.