Heuristics in Distributing Data and Parity with Distributed Hash Tables

by

Damon Earp

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
Dec. 11 2021

Keywords: Object Storage, Distributed Computing, Cloud Computing, CRUSH, CEPH

Abstract

We compare multiple methods of distributing data and error correcting code across distributed hash tables. We focus on the scaling of distributed hash tables and at which methods moved the least amount of data while maintaining an even distribution. A common technique is to use erasure coding and storing pieces of files on separate hardware. This approach makes placement of pieces dependent on earlier placements. We identify several rules that when applied to standard methods reduces the amount of data moved while scaling dramatically. Even though CRUSH [28] includes these heuristics we found that tweaking its approach allowed it to migrate less data when changing the cluster layout.

Acknowledgments

I would like to thank my advisor Dr. Xiao Qin for his patience and willingness to take me on. My friend and customer Dr. Navid Golpayegani of the Terrestrial Information Systems Laboratory (TISL) at NASA, for providing design input and resources which led me to here. Dr. Curt Tilmes (TISL), for his previous work which inspired this project. I'm thankful for the hard work and help Mr. Lawrence Sebald, University of Maryland Baltimore County (UMBC), and Mr. Jihad Ashkar, Science Systems and Applications Inc, provided to the project. A big thanks to Mr. Edward Masuoka and Mr. Robert Wolfe of TISL for taking a chance with this project and the supportive environment. I extend a thank you to the faculty and staff of graduate studies at UMBC, Naval Postgraduate School, College of Charleston, The Citadel, and Auburn University for the journey over the last decade. Finally I'd like to thank my wife Dr. Susan Earp for her loving support and our son Nathaniel Earp for keeping me sane even through a pandemic.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work highlights a set of techniques that minimizes data migration in decentralized object stores. We will demonstrate how these techniques improved a custom distribution algorithm as well as two prominent industrial standard methods: CRUSH [28] and Consistent Hashing [13].

Over the last couple decades decentralized object stores have gained popularity in the form of NoSQL databases, key/value stores, web caches, and general data archives. Their appeal includes the ability to easily scale, evenly spread load, ensure data availability and resilience, and run on cheap commodity hardware.

An object store is considered decentralized when the placement of data can be carried out without coordination [15]. Large scale centralized storage systems have single points of failure and bottlenecks when locating data. A popular approach to design a decentralized object store is to use a Distributed Hash Tables (DHT). DHTs are desirable as they are simple in design and fast at determining data placement.

To provide data resilience, decentralized object stores will store the data including parity information. There are two primary ways to provide parity information, replicas and error correcting codes like erasure coding. In a standard DHT all this information would be stored together with a single place. However to provide better availability and further increase resilience these systems will break up the data and parity information into even sized blocks and store each block on different hardware to minimize the effects of failures, data corruption, and system maintenance. When using replication each block is a copy of the data, and when using erasure coding the data is split up into multiple blocks and extra parity blocks are generated. Improving how these stores distribute blocks across a DHT is the focus of this work.

The placement of blocks affects three specific metrics of a storage system. How much data must move when new nodes are added or old nodes are removed from a store? Is the data and parity information evenly distributed? And how long does it take to calculate where to store each block? We will focus on the first two, as systems like Ceph [27] negate the cost of placement by distributing buckets, known as placement groups, instead of the data keys ahead of time.

In the rest of this chapter we will go into detail on what we are trying to solve 1.1, what our solution was 1.2, and what contributions we have made 1.3.

1.1 What we are Trying to Solve

We set out to find a better method of distributing data on our custom storage system that minimized data movement when scaling. We built a simple storage system that utilized a simple hashchain distribution algorithm discussed in 3.7. With 24 Petabytes of capacity and billions of files we noticed adding new nodes to the system was moving more data than we believed to be reasonable. Our investigation found that our naive algorithm distributed files in a way that caused a cascaded of movement. When a block is moved the probability subsequent blocks are moved is higher than it needs to be. Figure 1.1 illustrates how ensuring that blocks are placed on separate nodes causes a dependent relationship between all the blocks.

We began researching how existing systems handled distributing blocks across a DHT ensuring all blocks resided on unique nodes. We tried several modifications to our distribution algorithm and looked at how other systems handled collisions to reduce the likelyhood a block being moved would cause others to move as well.

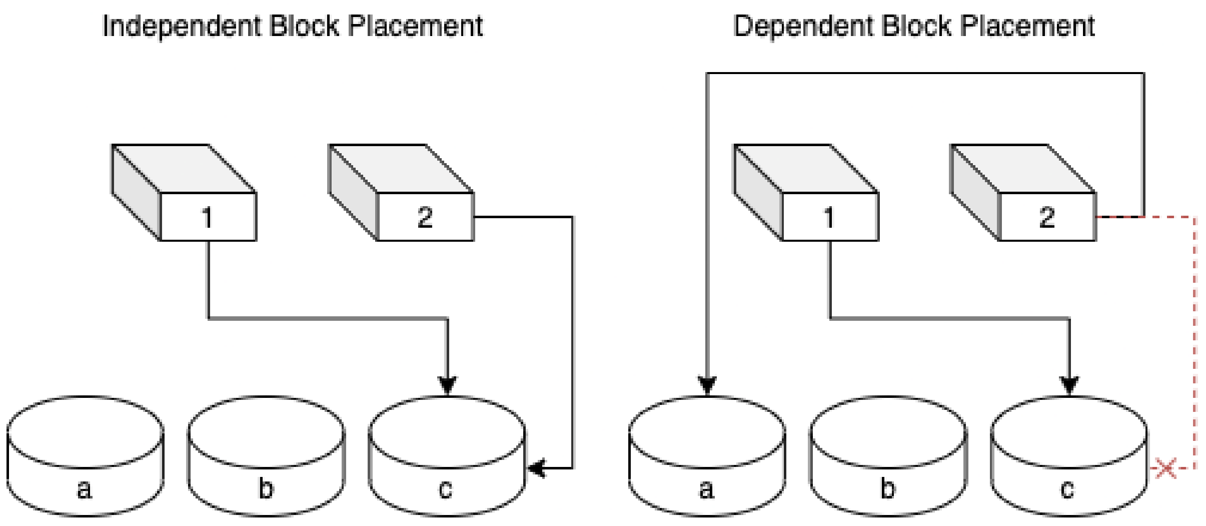

Figure 1.1: Independent vs Dependent Block Placement. To maximize Reliability and Availability of stored data, Distributed Object Stores will make sure that a node is not responsible for more than a single block. This restriction means that subsequent blocks are dependent on where previously placed blocks are located. When a block is mapped to a node that is already in use a collision occurs.

1.2 What was the solution

We discovered some simple techniques that greatly reduced the migration cost of our hashchain algorithm while still evenly distributing the data across the available storage hardware. We also found that applying these techniques to existing distribution algorithms, like CRUSH and consistent hashing, we were able to further minimize data movement when scaling a cluster.

A collision happens when two or more blocks want to be placed on the same node. How the distribution method deals with resolving these collisions greatly affects how much data is moved during migrations. In general to minimize data movement distribution methods should:

- Not try to place all blocks in a single pass. By deferring blocks that collide with others to future passes one minimizes the chance a single block move will cause multiple subsequent movements, a cascade.

- Not place blocks relative to previous placements. Rings are a popular topology for DHTs, however as we'll see in 5.5 simply placing blocks at offsets after the initial placement leads to many unnecessary moves.

- Encode information about the placement. We found altering the set of nodes to consider after each pass improved CRUSH (4.2), while altering the key of a block each pass improved our hashchain and consistent hashing methods.

1.3 Our Contribution

To understand why our distribution method was moving so much data, we implemented a simulator. Over time we continued to try new things, and added other methods to the system. Today we have a framework that allows for fast prototyping of a distribution method and a simple way to compare it to other methods.

The outcome of this research is we:

- Defined a few techniques 1.2 that when applied to naive distribution algorithms, greatly reduces the cost of data migration.

- Implemented a simulator C.2 which allows fast prototyping of methods and testing. The input includes file names and their sizes, how many nodes to start and end with, how many buckets to distribute across the cluster, and the erasure coding settings being used. The output is a raw dump of where a block is located before and after a migration and size of the block in bytes.

- Wrote an analysis script C.4 to take the output of our simulator and provide a statistical analysis of the data migration. The script provides information on how much data moved, how many blocks moved, and an analysis of how evenly distributed all the data is across the cluster.

- Implemented and tested over 20 different methods to distribute blocks. The methods fall under four basic types: Hashchain 3.7, Ring and Consistent hashing 3.5, CRUSH 3.6, and Shuffle A.1.6. The Hashchain family maps blocks to a set of nodes where each node has assigned hash ranges. Ring and Consistent Hashing methods arrange the nodes in a circle mapping blocks around the same ring. CRUSH using the straw2 strategy is akin to the nodes drawing straws to decide who gets a block. Finally shuffle plays with a standard array shuffle algorithm to create a list of nodes to store each block on.

- To test CRUSH and its derivatives we modified Ceph's libcrush [26] library and provide a C++ application C.3 to use libcrush to test migrations and provide the same output as our simulator.

- We provide a patch to libcrush's ChooseN algorithm, see Appendix B, that decreases data movement of the CRUSH algorithm when using Straw2 and erasure coding 4.2.

Chapter 2
Related Work

There has been an increased focus on distributed storage systems over the past couple decades. Websites serving billions of users have caching and archival needs which must scale, provide constant access to data, and handle failures gracefully. The rise of cloud computing with the promise of unlimited user storage is now possible. There is a lot of research on different algorithms to store and retrieve data that is spread across multiple servers. We will look at several enterprise systems and research projects that aim to build better distribution systems.

2.1 CRUSH

Proposed back in 2006 CRUSH [28] is the backing distribution algorithm of Ceph, a leading industry object storage system. CRUSH was developed to be scalable and minimize unnecessary data movement when adding and removing hardware from the cluster. It is derived from the RUSH [11] family of distribution algorithms, and offers a flexible and efficient way of storing data in a weighted hardware hierarchy. One of the unique contributions of this work is the notion of drawing straws based off of the id and weight of a component. Most systems distribute the keys by mapping the key to a host, with straw hosts bid on a key instead. CRUSH was designed with storage of replicas and erasure encoded blocks in mind.

2.2 Ceph

Currently managed by Red Hat, Ceph [27] is an enterprise ready object store with proven record for storing hundreds of petabytes of data. The project has a library implementation of CRUSH [28] that we were able to use to compare CRUSH's data movement efficiency with other algorithms. A modification to CRUSH's straw method was implemented in Ceph [25], today straw2 is the default distribution method. While the distribution of data is decentralized, Ceph does have several manager services that administers must run multiple copies of to mitigate the impact of failures.

2.3 DynamoDB

Amazon's DynamoDB [9] is a decentralized key value store designed for high availability where small outages have large financial consequences to a large company. Dynamo uses consistent hashing to distribute the storage nodes, data, and parity around a ring. To address limitations with consistent hashing, and to provide support for nodes of differing capacities, DyanmoDB maps nodes multiple times called virtual hosts. Hashing is used to find where to store a given key and a preferred list is generated by skipping virtual hosts that point to already used nodes.

2.4 Cassandra

Apache Cassandra [15] is a key value store developed at Facebook with an author of DynamoDB. It was designed to run on hundreds of commodity hardware systems and even across multiple data centers. To address the limitations of consistent hashing Cassandra will move lightly loaded nodes to more optimal placements, increasing the load of the node and reducing the load of others. This differs from DynamoDB which does not move nodes but maps them several times.

2.5 Attributed Consistent Hashing

The Attributed Consistent Hashing paper [29], focused on comparing CRUSH, traditional Consistent Hashing, and their Attributed Consistent Hashing algorithms on hardware using a mixture of both spinning and solid state storage devices. This work provided some insight into the characteristics of Consistent Hashing versus CRUSH however it primarily focused on I/O Bandwidth and Key Placement runtime. While they did compare data on data movement during a migration, they only compared the two consistent hashing methods.

2.6 ZHT: Zero-Hop Distributed Hash Table

Another consistent hashing system is ZHT [16]. Instead of using a hashing function to distribute nodes around the ring, ZHT adds nodes by placing them as neighbors to heavily loaded nodes. The entire hash space is broken up into a static number of even sized partitions, each node then is assigned one or more partitions to manage. Another eccentricity of ZHT is replicas are not placed in a clockwise or counter-clockwise manner, instead they are distributed by hoping back and forth in both directions.

2.7 ECHash

ECHash [7] is a consistent hashing system that is designed to make storage of data and erasure coding parity more efficient while scaling. The authors noticed read penalties when adding and removing nodes as recovering data with erasure coding has a computational cost. Unlike other consistent hashing systems ECHash makes use of multiple hash rings. When coupled with their fragmented erasure coding scheme, FragEC, they were able to minimize read penalties when scaling a cluster.

2.8 Distributed File Systems

Distributed file systems aim to provide users a familiar experience to traditional local versions. Benefits include access to data on any networked computer and automatic data resilience. By emulating the features of local file systems the distributed versions tend to have centralized infrastructure to handle features not supported by object stores, file locks are a prime example. Systems like Lustre [5] and the Hadoop Distributed File System (HDFS) [18] are heavily dependent on their name or metadata server for locating and storing data with parity information. Disabling these services halts all operations on their cluster.

A decentralized distributed file system is GlusterFS [10]. Each node in the cluster exports one or more bricks, and those bricks are combined into volumes. Rules on the volumes dictate how files are stored. Replicated volumes will copy the data to every brick, while striping volumes spread the data across all bricks without parity. These building blocks have allowed GlusterFS the ability to work without the need for a centralized metadata service.

Chapter 3

Background

Data availability and resilience are important properties of distributed storage systems. In Section 3.1 we discuss what each property means and how it is achieved. Section 3.2 we will go over replication and erasure coding which supply parity information allowing high levels of availability and resilience. We will quickly discuss hashing and what properties decentralized storage systems look for in a hash function in Section 3.3. We will go over Distributed Hash Tables (DHT) 3.4 and several types of distribution methods built on top of DHTs including Consistent Hashing 3.5, CRUSH 3.6 and Hashchains 3.7. Finally in Section 3.8 we discuss how distributing buckets that we hash files into has several benefits.

3.1 Data Properties

The promise of distributed data stores is increased availability, easier maintainability, and stronger resilience and reliability [2]. To accomplish these goals distributed stores spread their load across the whole cluster, and remove single points of failure using error correcting codes or replication.

Availability of data is one of the key features of distributed object stores. These systems are designed to minimize the impact of failures to the end user. Availability is impacted primarily by system maintenance and hardware failures. System administrators need to reboot nodes as well as upgrade software, all of which could potentially lead to a user not being able to access their data. Furthermore a node becoming unresponsive due to network or power outages, and even hardware component failures must be mitigated.

Data Resilience is the measure of how likely data will be lost due to hardware failures or corruption. The main hardware failure affecting resilience are hard drives, once a drive is lost so is all the data stored on it. Data corruption can happen due to bad Random Access Memory and bit-rot. By storing parity information these systems can automatically generate and restore missing or corrupt data.

3.2 Replication and Erasure Coding

To facilitate all the above properties distributed data stores also store parity information in the form of duplicates or error correcting codes. Coupled with a distribution algorithm that stores data on many different systems, or even racks, or data centers, minimizes the impact of system maintenance, hardware failures, and data corruption. Due to benefits and trade offs of both methods hybrid systems can be used.

Replication is the most straight forward approach. When storing data the system duplicates it a configurable number of times and stores the copies on separate nodes. As long as the data on one of the nodes is available, users can access their data. The benefits of replication include decreased disk and network access to retrieve requested data. However the storage overhead to provide similar resilience and reliability of erasure coding is vastly greater. As a general rule replication systems always hold at least three copies of the data, enterprise systems like Ceph and HDFS default to three, as the probability of two hardware failures making files inaccessible is too high. Three replicas means a system will have 200% overhead which is costly, to store 8 petabytes one must have at least 24 petabytes of storage.

Erasure coding provides an alternative to replication. By using error correcting codes one can split the data into a set number of equal parts and encode a configurable number of extra parts. For example one can split a file into three equal sized pieces, and then generate two extra parts. The data is now represented by five equal sized pieces, and any missing combination of one or two parts can be regenerated with the others. If the five pieces are stored on separate nodes, it has the same availability as a replication of three but at a storage overhead of 66.67% instead of 200%. The reduced storage overhead comes at the cost of more network and disk accesses, and a computational cost when regenerating inaccessible pieces.

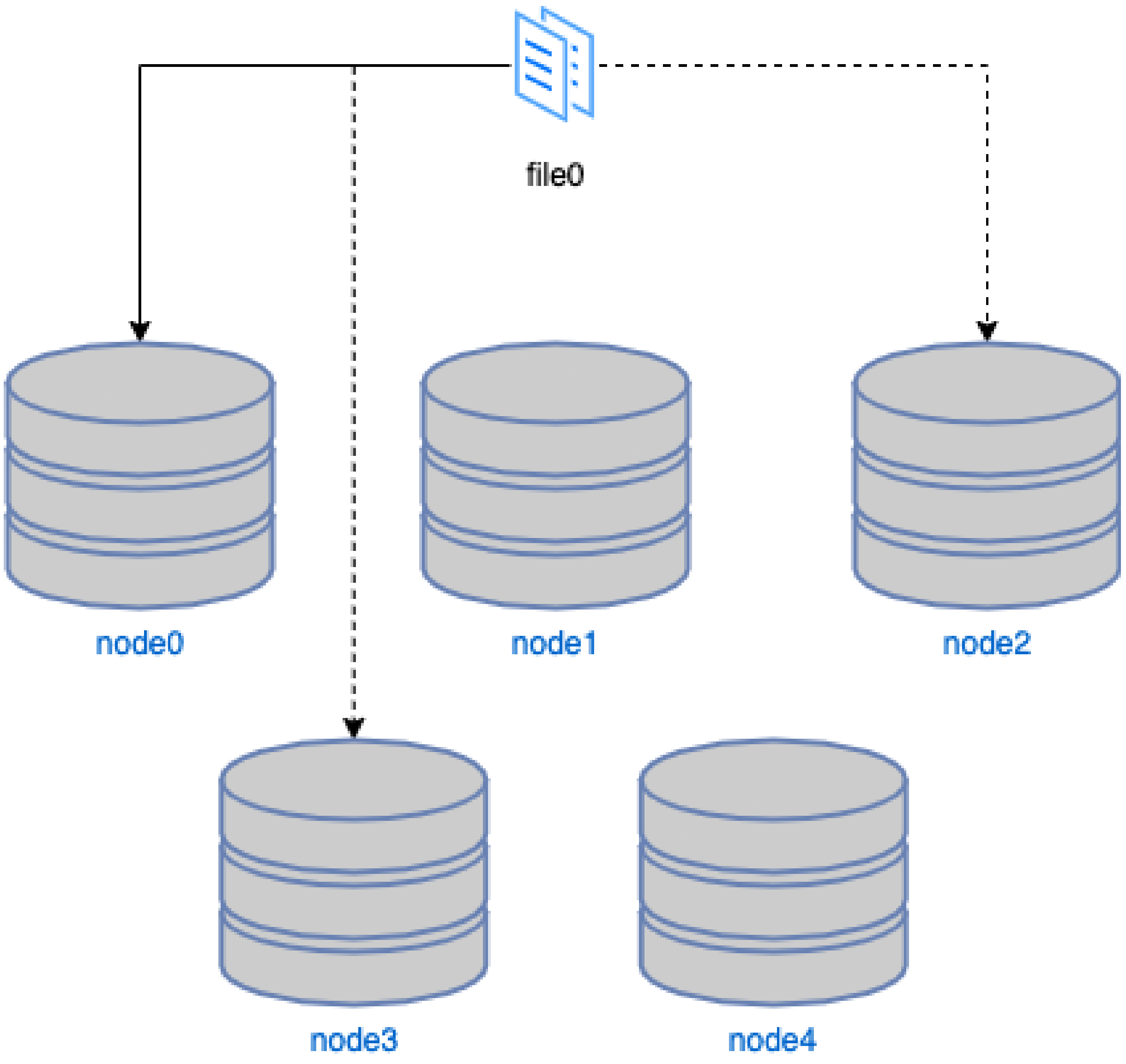

Figure 3.1: Replication: Data availability and resilience can be increased by replicating data. To replicate the data is stored in its entirety on multiple nodes. Several major systems, such as Ceph and HDFS, default to at least three copies. Replication has a low computational cost to recover data that was stored on failed hardware, however the storage overhead is very high. A cluster that is expected to store 8 petabytes must have a minimum capacity of 24 petabytes to support the two extra copies.

Production systems can store erasure encoded data across 20 or more machines, on such a cluster to access that data a user must communicate with 20+ nodes.

3.3 Hashing

To distribute files in a decentralized cluster a popular approach is using a Distributed Hash Table 3.4. Like traditional hash tables a key is fed into a hashing function and the resulting value is used to place the result. Hash functions have the following properties: their outputs are evenly distributed across the range of all possible outputs, they are one way functions, and they are collision resistant.

The key property we are interested in is uniformity. A hash function that spreads its input across all possible outputs evenly will allow Distributed Hash Tables to distribute the data evenly over their nodes. If a group of similarly named files are very large, a uniform hash function will ensure that they are not placed together.

A one way function is a procedure where the input cannot be derived from the output. This is a very important property for cryptography, for example the storage of the hash of a password should not leak what the password actually is. In terms of distributing keys across a hash table this property is not important. however the majority of hash functions have this property.

Collision resistance states that it should be hard to find two inputs that generate the same output. This is of primary interest to cryptographic systems.

While cryptographic hash functions can be used to distribute files evenly, their large hash space is unneeded, they do not provide better uniformity, and they are slow compared to non-cryptographic alternatives. Popular non-cryptographic hash functions include Jenkins, MurmurHash3, and City Hash.

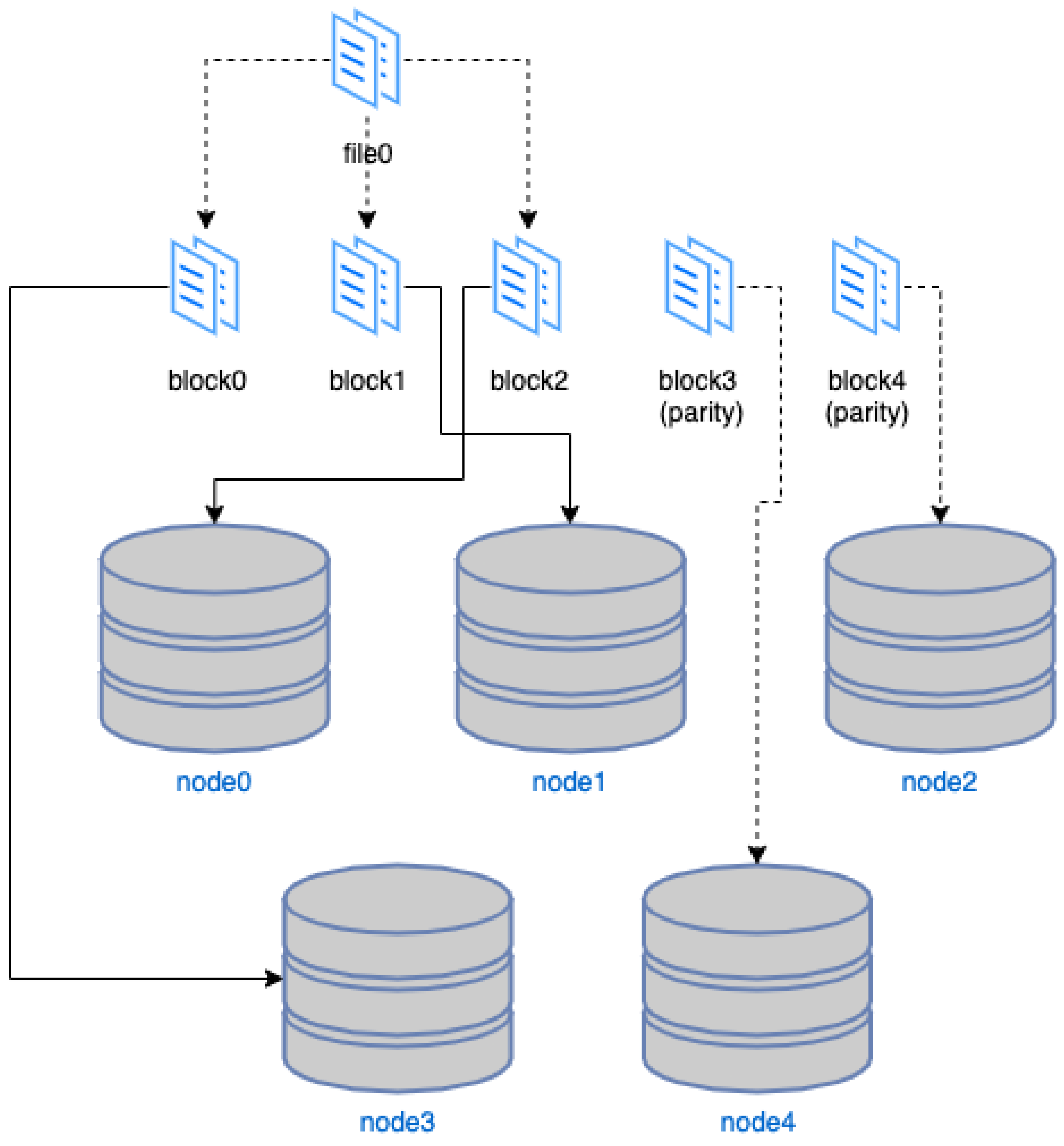

Figure 3.2: Erasure Coding: An alternative to replication, erasure coding is an error correcting coding that can provide the same level of availability and resilience at a fraction of the storage overhead. In this example the file is broken up into three equal sized blocks. Two extra blocks are computed from the first three. Armed with five total blocks as long as three are accessible and correct any combination of two blocks can be recalculated. With this scheme the storage overhead is 66%, far less than the 200% overhead a system that stores three copies of all the data. Note it is very easy to lower this overhead further. Splitting an object into 17 pieces and generating three parity blocks reduces this overhead to $< 20\%$.

3.4 Distributed Hash Tables

The heart of many distributed storage and caching systems is a Distributed Hash Table [4]. All the methods described and examined in this paper are DHTs. What sets these methods apart is how they distribute parity information across unique nodes. DHTs can scale well [4] and provide an easy method to work decentralized. A simple, but naive, implementation of a DHT is to have an ordered list of nodes. To place a key value pair one simply computes the index of the owning node like so: $index = HASH(key) \bmod |nodes|$. Over the years there have been a plethora of systems backed by DHTs. Big name projects include Ceph [27], DynamoDB [9], HDFS [18], and Apache Cassandra [15].

3.5 Consistent Hashing

Consistent Hashing was developed to counter the downsides of simple ring method DHTs. By arranging nodes in a set order and treating them as a ring it is very quick to find what node a key should be placed however it ties the location of a key to the size of the cluster. This dependent relationship causes the remapping of keys all over the ring leading to a lot of unnecessary data movement when scaling the cluster and motivated to the Consistent Hashing approach [13].

Instead of arranging the nodes in a ring then mapping keys to the resulting circle, consistent hashing maps both nodes and the keys. Nodes can be deterministically assigned on the circle using a hash function. To store a key value pair one hashes the key and searches for the nearest node with a hash greater than or equal to the hash of the key. This approach allows the addition of new nodes and removal of old ones to only affect keys that hash close to the node. In this base form consistent hashing helps reduce hot spots produced by resizing a cluster.

A major shortcoming of consistent hashing is the hash space a node is responsible for is assigned randomly. The hash of a node could place it on the circle such that the previous

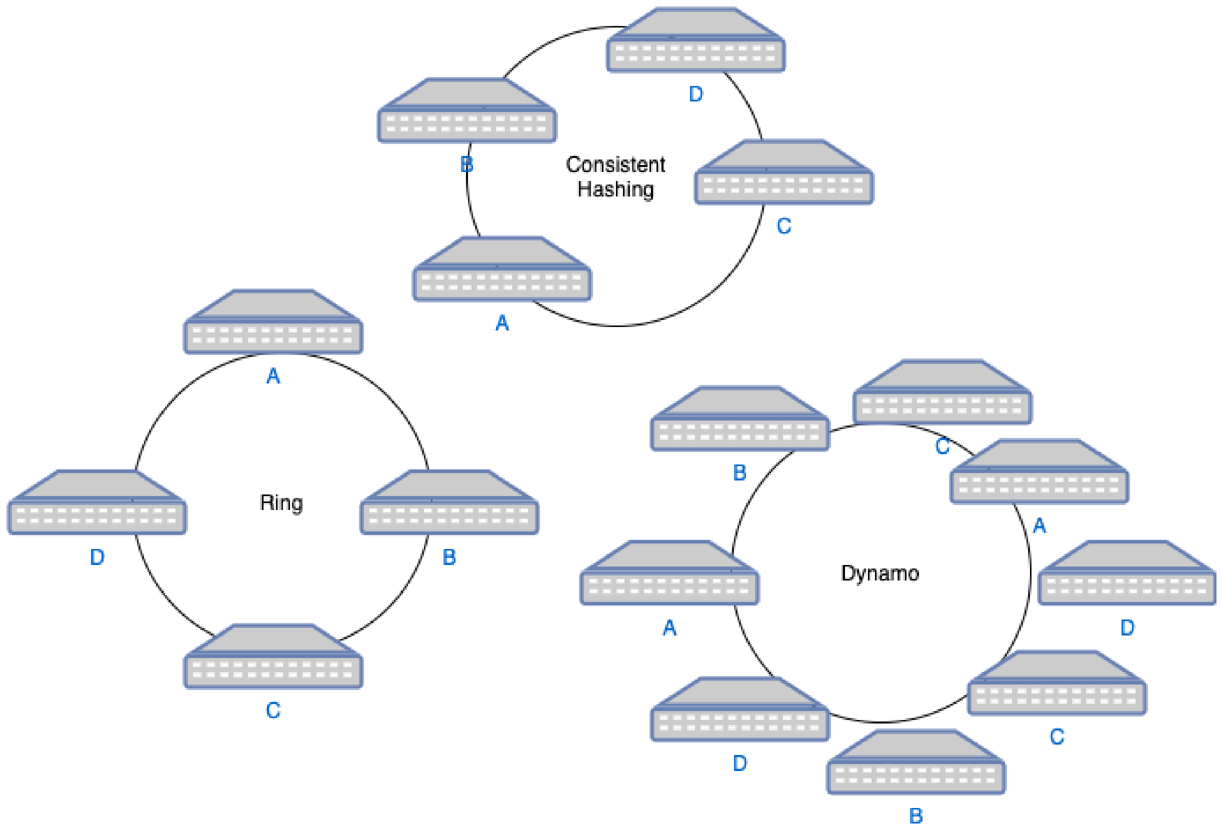

Figure 3.3: The ring illustration shows how nodes A through D would be arranged in a ring. Each node will be equally spaced around the ring but keys will only ever be hashed to one of four values. With Consistent Hashing we see that the four nodes are mapped to random locations around the circle and the spacing between the nodes is not even. When a key is mapped to the circle and assigned to the next node counter-clockwise we can see that Node C will be responsible for the most keys. Finally The Dynamo Circle represents their take on Consistent Hashing where each node is mapped twice to the circle. By placing double the number of nodes there is less variance in the number of keys a node is responsible for.

node is very far away, or very close. This discrepancy in hash space sizes will lead to an imbalance in the cluster as keys are more likely to hash to a node with a larger hash space than a smaller one. To address this systems like DynamoDB [9] will place nodes on rings multiple times. The more nodes placed the less variance between hash space sizes. Another approach used, by Cassandra, is to map all the nodes to the circle to reduce the load on heavily assigned ones [15]. Figure 3.3 is a simple illustration of how Ring, Consistent Hashing, and Dynamo DB's consistent hashing approach would distribute four nodes.

To store replica or parity data on different nodes each system works differently. Dynamo and Cassandra both map all the nodes to the circle, finds the closest node, and $n - 1$ subsequent unique nodes to create a preferred list [9] [15]. Meanwhile ECHash uses multiple circles [7] to determine where each block should be placed. Finally ZHT places blocks on either side of the node responsible for the key [16].

3.6 CRUSH

The CRUSH algorithm is a generic way of describing one's infrastructure and providing explicit rules on how to store data. With CRUSH one can replicate a file across multiple data centers, or one can use erasure coding to store files entirely on a single rack but each block on a different node. This flexibility is the backbone of systems like Ceph [27] and Twitter's Blobstore [12]. The CRUSH paper describes multiple distribution strategies with straw being the most interesting, in Ceph the default is straw2 [25] a derivative of straw. Unlike DHTs that arrange nodes in a ring where a key is mapped to hardware, CRUSH with straw allows hardware to bid on who will be responsible for the key. As outlined in their paper, hardware draws straws deterministically and the largest value wins, this is illustrated in Figure 3.4.

What makes straw so compelling is that it is the most flexible when scaling a cluster. CRUSH does outline the uniform, list, and tree strategies which perform better than straw in certain situations. For example a JBOD has a fixed number of disks, and the node will more likely be replaced than upgraded to hold more disks. Since this is a static count of disks, straw is overkill and uniform would be a valid choice. However using uniform strategy to distribute data across nodes will move a lot more data when scaling than a system using straw [28]. The downside to straw is that compared to other distribution methods it is slow. If we have a cluster with 100 nodes in it the consistent hash approach only needs to do a single hash and log(n) search to place a file, CRUSH with straw has to perform n hashes.

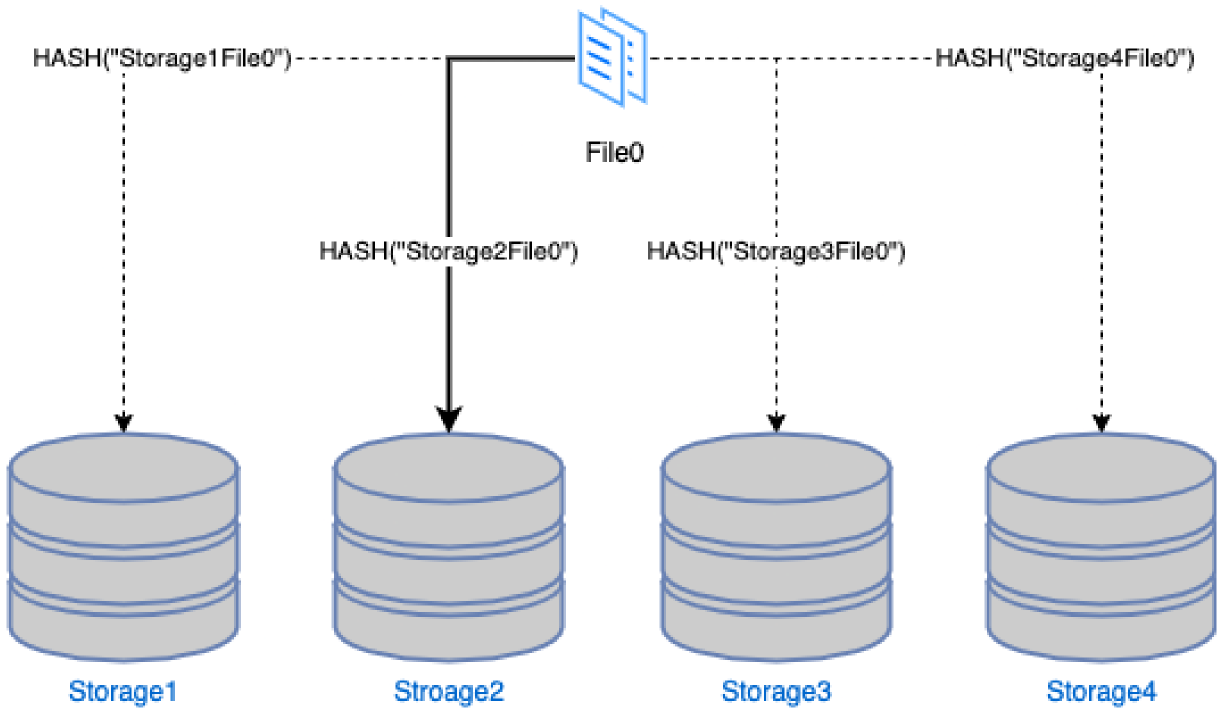

Figure 3.4: CRUSH with straw strategy is akin to all the nodes in the system drawing straws to see which one is responsible for a given key. Each node has a unique identifier that when hashed with the key provides a numeric straw, the largest value wins. Here Storage2 when paired with the filename File0 hashes to the largest value and is responsible for the key "File0".

To handle clusters with heterogeneous hardware every single node CRUSH considers has a weight associated with it. This weight is then applied to the drawn straw allowing nodes with larger capacities to have a higher chance of winning the draw. A basic setup is each disk gets a weight of 1.0 and each level above gets a weight of the sum of all children. A 90 disk JBOD with all disks weighted at 1.0 would have a weight 90.0. This allows smaller nodes to not get overwhelmed. The primary difference between straw and straw2 is how the weights are applied.

Since CRUSH with straw strategy doesn't have a simple ring to go around placing replicas or parity blocks it must draw straws for each block. To ensure parity data is stored on separate nodes, CRUSH makes a multi-pass approach, where the first pass places all blocks it can, and future passes revisit blocks that collided with another block.

3.7 Hashchain

In our first foray into Distributed Hash Tables we choose to map ranges of a 32 bit hash space directly to nodes in the cluster. The size of the range represents the capacity of a node, for example a uniform cluster assigns each node an equal sized range. To place a value one hashes the key and performs a log(n) search for the range which contains that hash. The primary downside to this approach is that it leads to fragmentation. As nodes are added and removed over time the ranges must be split and merged, adding nodes means stealing an equal number of hashes from all nodes to even the load. Compounding to this we never got to automating the assignments, meaning all cluster changes were manual endeavors. This alone has lead us to be jealous of both consistent hashing 3.5 and CRUSH 3.6.

To distribute parity information we used a chain of hashes A.1.1. The hash of the key is the beginning of this chain, all subsequent blocks are placed by hashing the previous hash. When the hash of a block causes a collision, we continue hashing until the block is mapped to an unassigned node. This process is illustrated in Figure 3.5. This naive approach moved so much data that it was the motivation for this work.

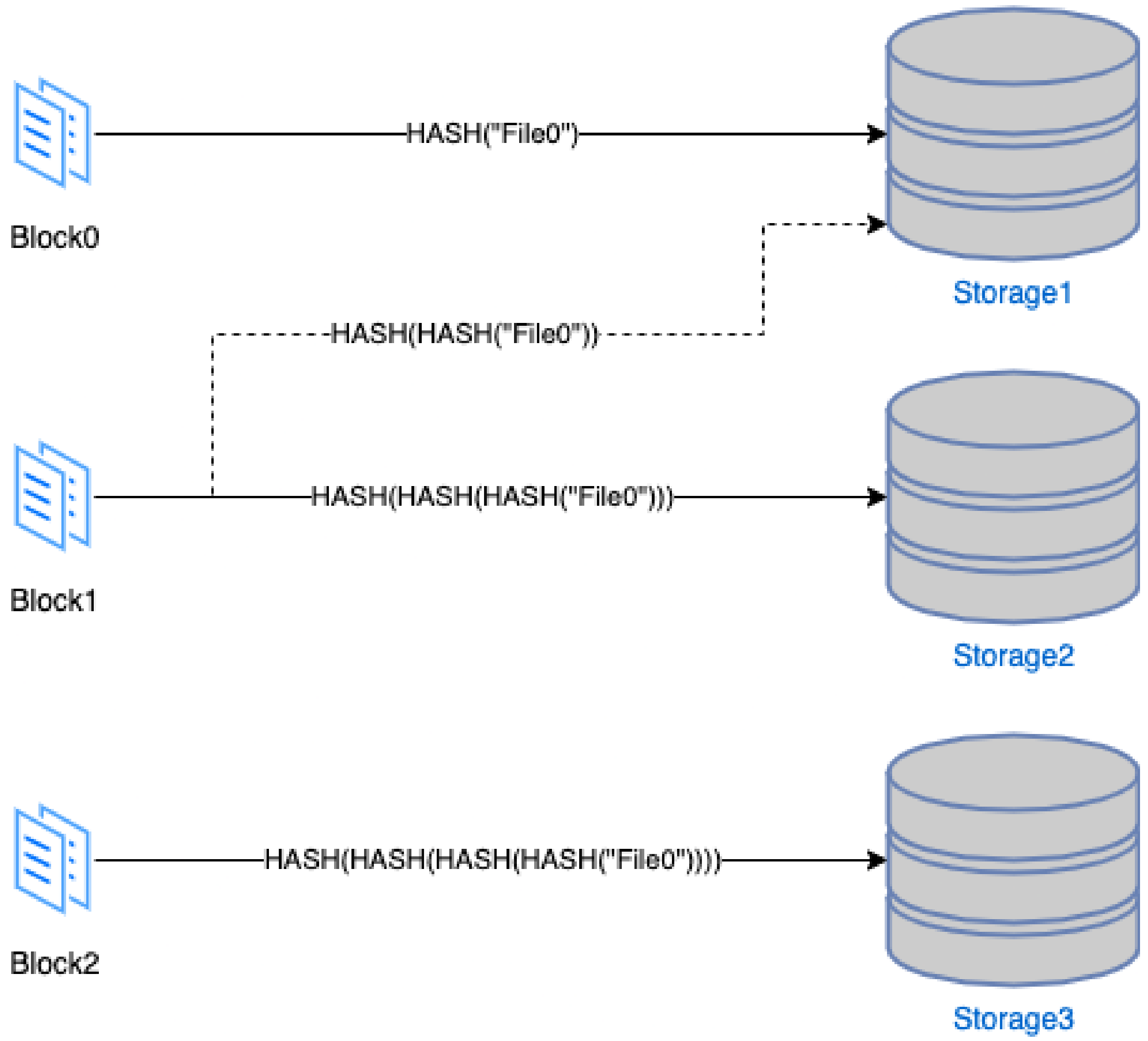

Figure 3.5: A simple Hashchain placement of a file across three storage systems. ”File0” is stored as the blocks and each block uses the hash of a the previous placement. The second block has a collision with the first one therefore we hash this value again till we find an unused node.

While researching other distribution methods and ways to improve Hashchain we found a way to build the chains such that the migration performance is on par with CRUSH's straw strategy. Called Hashchain', see A.1.2, works by making two changes, first on collision the colliding block is skipped to be placed in a later pass, and second hashes that collide spawn separate chains.

3.8 Distributing Buckets Not Keys

Consistent Hashing 3.5, CRUSH 3.6, and Hashchains 3.7 all use the hash of a key to distribute the value and parity information over a storage cluster. Systems like Ceph noticed that CRUSH with straw is relatively slow when compared to the simple hash and modulo approach of a ring distribution. To reduce the overhead of finding where a file is stored and group files together into fewer units for easier management Ceph created Placement Groups [27]. On setup these systems use their backing algorithm to distribute all the buckets and cache the nodes responsible for each bucket. This approach allows placement of files to go from $O(n)$ for straw to $O(1)$. Mapping files to buckets instead of directly to nodes means files will never move to another bucket even as a cluster grows and shrinks.

Chapter 4

CRUSH' - A Scalable and Decentralized Placement of Replicated Data

We propose a change to how the CRUSH algorithm handles collisions. A collision happens when distributing blocks multiple blocks are to be stored on the same node. There are many ways to handle collisions and not all are equal.

We will go over how CRUSH works to distribute objects that use erasure coding 4.1. We will then discus CRUSH' 4.2 which modifies CRUSH minimizing data movement when scaling a cluster. Finally we will discuss the modifications to libcrush 4.3 written and maintained by the Ceph open source project.

4.1 CRUSH

CRUSH [28]. When distributing replicated data there is no difference between blocks stored at index 0 or at index n. This differs from data using erasure coding methods as every single block differs and their ordering is important. The CRUSH algorithm handles this case by placing each block in an index that is a multiple of the total number of blocks. For example if block 2 is to be stored on the same node as block 1, to resolve this collision CRUSH calculates a new hash using index $2 + n$ then $2 + 2n$ until the block is placed on an unused node. The pseudo code in 4.1 illustrates this idea.

Previous methods, like consistent hashing and hashchain, simply skip indexes that lead to collisions. This technique allows the algorithm to place all the blocks in a single pass just by incrementing the index. CRUSH could have done this by altering the block index until a collision is avoided, however the authors chose to place colliding blocks in future passes. Each pass CRUSH places all the blocks that do not collide, this technique reduces

Algorithm 1 Pseudo code illustrating how CRUSH's ChooseN algorithm would select n distinct nodes to store erasure coded blocks.

```
procedure ChooseN(nodes, n, key)
    $result \leftarrow [1..n]$
    $used \leftarrow \{\}$
    $revisit \leftarrow \{1..n\}$
    $r \leftarrow 0$
    while $|revisit| > 0$ do
        for $i \leftarrow 1$ upto $n$ do
            if $i \in revisit$ then
                $h \leftarrow HASH(key, i * r)$
                $node \leftarrow STRAW2(nodes, h)$
                if $node \notin used$ then
                    $used \leftarrow used \cup \{node\}$
                    $revisit \leftarrow revisit - \{i\}$
                    $result[i] \leftarrow node$
                end if
            end if
        end for
        $r \leftarrow r + 1$
    end while
end procedure
```

the likelyhood a block being moved when scaling will cause other blocks in the object to move as well.

While searching for solutions to our distribution algorithm we found that this multi-pass approach is quite effective in reducing the amount of data moved when resizing the cluster. However we also found other alterations which improved our algorithm further. After comparing our improved algorithm to CRUSH we decided to see what would happen if we tweaked CRUSH in different ways.

4.2 CRUSH'

CRUSH handles collisions by deferring placement to future passes and altering the blocks index by a factor of n 4.1. We applied the multi-pass technique and several other tweaks to our hashchain method and found we were equaling or beating crush in moving the least amount of data when scaling. One thing we noticed is every pass CRUSH considers all the nodes, including the nodes that are already in use. At first glance this seems inefficient, more hashes every pass and some passes could end up not placing a single block. We decided to see if playing with the set of nodes would affect how much data is migrated when scaling.

We came up with two possible ways to encode information about used nodes each pass. One approach is to re-weight nodes based off of their availability, for example after each pass increase unused node weights by a factor. The other idea is to remove all the used nodes at the end of each pass.

Unfortunately altering node weights did not provide any benefits. In CRUSH the weight of a node is a multiplier to the a nodes straw length during a drawing. The larger the weight the greater chance a short straw drawn will be transformed into the longest straw. This mechanism allows CRUSH to encode nodes of different capacities, as it will increase the number of wins of a large capacity node over smaller ones. After each iteration one could reduce the weight of nodes that have been selected, or increase the weight of unused nodes. Our initial tests found these alterations performed worse than the unaltered CRUSH.

Although altering weights did not provide any benefit, removing used nodes after each pass did. We modified CRUSH to keep track of all nodes that had blocks assigned to them in a pass. We then removed these nodes from the straw drawing of blocks that encountered a collision on the first pass. The final change removed the altering of the block index while drawing straws in subsequent passes. The pseudocode at 4.2 illustrates this approach. These modifications caused a decrease in amount of data moved while maintaining a uniform distribution of the data.

As noted above we removed altering the block index each pass as CRUSH does. This seemingly insignificant change is required for CRUSH' otherwise the algorithm performs worse than the original CRUSH. When CRUSH runs into a collision it simply iterates the block id to the next position to consider while continuing to consider all nodes. We found that using CRUSH's $i + r$ in the hash calculation while only considering available nodes provided no benefit and moved more data.

Algorithm 2 After each pass, all nodes that are in use are removed from the set of available nodes.

```
procedure ChooseN($nodes$, $n$, $key$)
    $result \leftarrow [1..n]$
    $used \leftarrow \{\}$
    $revisit \leftarrow \{1..n\}$
    while $|revisit| > 0$ do
        $available \leftarrow nodes - used$
        for $i \leftarrow 1$ upto $n$ do
            if $i \in revisit$ then
                $h \leftarrow HASH(key, i)$
                $node \leftarrow STRAW2(available, h)$
                if $node \notin used$ then
                    $used \leftarrow used \cup \{node\}$
                    $revisit \leftarrow revisit - \{i\}$
                    $result[i] \leftarrow node$
                end if
            end if
        end for
    end while
end procedure
```

4.3 libcrush Changes

To test modifications to the CRUSH algorithm, we were able to utilize Ceph's Open Source libcrush [26] implementation. Since we are focusing on the efficiency of data movement in storage systems using erasure coding we altered the *crush_choose_indep* function ignoring the replication specific $firstn$ counterpart. We implemented a simple *struct* $crush_bucket$ duplication function and used a fixed length array to keep track of items that were used. We provide the patch file to crush/mapper.c in Appendix B.

Chapter 5

Experimental Results

With a simple simulator and a database of billions of filenames and sizes we were able to test all methods in very specific situations. We have provided a small set of experiments to highlight how well each method works.

In this chapter we will go over the simulator 5.1, the common setup of all the experiments 5.2, and the different methods used to distribute files 5.3. Experiment 1 5.4 will compare how all the methods perform given the situation that made us look for other ways to distribute files. The second experiment 5.5 looks to see if the performance of each method is tied to how many nodes are added to the system, does method A work better when growing a cluster 4% while method B works best when growing a cluster 60%. The final experiment 5.6 tracks data movement across a cluster which grows, shrinks, and adds new files.

5.1 Simulator

We built a couple programs to simulate all our ideas on how to improve data movement performance. To test CRUSH and any derivatives we wrote a simple C++ program and edited a local copy of libcrush. All the other methods are implemented in a Perl program for decent performance and fast prototyping. Each program reads in a set of files and their sizes and outputs how a file would be distributed before and after node additions or subtractions.

Since both programs output in the same format we wrote a simple Perl script to analyze the results. We calculated statistics on how much data moved given the cluster change and what the standard deviation of data stored per node is. These statistics with the fast prototyping allowed us to try dozens of alterations to each of the algorithms we used in our experiments. The source code for our simulator can be found in Appendix C.

5.2 Setup

For our testing we focused on a cluster of uniform nodes. This restriction allowed us to isolate how well a method avoids unnecessary data movement when scaling a cluster.

CRUSH is a general algorithm that can work on various infrastructure designs and custom heuristics. A user has the freedom to setup a cluster where there are no restrictions on where blocks can be stored, multiple blocks can live on the same hard drive, or a user can dictate blocks must be distributed so only one lives on a rack of servers. Additionally CRUSH uses a weighted system to help distribute data evenly across systems with varying capacities.

Consistent hashing methods [13] were designed to improve on ring based DHTs. Instead of mapping keys to nodes, one maps both keys and nodes to a circle. The node with a hash greater than or equal to that of the key is the node responsible for that key. Like CRUSH and Hashchain we provide an implementation of Consistent A.1.4 and Consistent' A.1.5 a derivative which utilizes the rules we have learned during this research. Consistent is modled after the algorithm laid out in DynamoDB [9]. The key feature of this algorithm is to map nodes multiple times to evenly distribute the load to all nodes. To get the standard deviation of data stored per node close to the other methods we placed all nodes 100 times around a ring. The larger this factor the smaller the standard deviation of data stored per node is. The factor also affects search times of lookups, binary search is a practical option to offset this cost. One could easily add nodes 10,000 times to continue to reduce the standard deviation, however we choose to keep our tests smaller. At a factor of 100 a 50 node cluster has 5,000 entries to search through.

5.3 Methods

In our experiments we focus primarily on CRUSH (4.1) and our updated CRUSH' (4.2). We also compare other methods used to distribute files including Hashchain (A.1.1) our

	Data Moved	**Bytes**	**Blocks**	**Std Deviation**
CRUSH'	45.47%	4.51 TiB	9,107,593	7.21 GiB
CRUSH	47.53%	4.72 TiB	9,501,073	7.51 GiB
Hashchain'	47.30%	4.7 TiB	9,492,599	7.05 GiB
Hashchain	72.92%	7.24 TiB	14,596,048	7.84 GiB
Consistent'	51.55%	5.12 TiB	10,342,606	14.36 GiB
Consistent	89.95%	8.93 TiB	17,965,282	16.27 GiB
Ring	62.76%	6.23 TiB	12,534,389	8.47 GiB
Shuffle'	49.17%	4.88 TiB	9,812,080	58.1 GiB

Table 5.1: Results from our first experiment. We see that when going from 20 nodes to 29, our modified CRUSH algorithm, CRUSH', was the most efficient at moving data and provided the second best even distribution of that data. Hashchain, our initial algorithm,

naive attempt, Hashchain' (A.1.2) our improved Hashchain, Consistent (A.1.4) the consistent hashing approach used in DyanmoDB, Consistent' (A.1.5) an improved version of Consistent, Ring (A.1.3) a basic ring topology, and Shuffle' (A.1.6) which plays with array shuffling. There are some differences in hash functions between the methods. CRUSH and CRUSH' both use the Jenkins hash function to map filenames to buckets. All other methods compared use MurmurHash3.

5.4 Experiment 1: Simple Increase

For the first experiment we used the same configuration of our production cluster. We had a batch of new servers come online and wanted to increase the space. Each node had 90 storage drives of 10 Terabytes each.

We started with 20 nodes growing the cluster to 29 nodes. We used an erasure coding of 16/4, all files are split into 16 blocks and 4 extra blocks were generated. While we have billions of files in our system we used one million random files with their size as input. With the overhead of the erasure coding we analyzed a cluster's management of 9.93 TiB of data. Table 5.1 shows the results of distributing 1024 buckets and placing the input files into those buckets.

From 5.1 we see that CRUSH provides fairly efficient data migration while maintaining a small standard deviation on space usage. It was quite a shock to see our naive hashchain method perform so poorly, but our improved method hashchain' not only moved less data than CRUSH but also maintained a more even distribution. CRUSH', our alteration to CRUSH, moved 200 GiB or 400k blocks less than CRUSH and provided similar data distribution to hashchain' as well as CRUSH.

5.5 Experiment 2: Scaling Up

In Section 5.4 we grew the cluster by 45%. In our second experiment we wanted to see if this advantage is dependent on the number of nodes being added to the system. We started with a 25 node cluster and compared the different methods when adding 1 up-to 25 nodes into the cluster. We distributed 1024 buckets to put files into and each file was erasure coded with 20 and 5 erasure blocks. A total of 2.8 TiB were distributed from a little over 87,000 files. Table 5.2 shows the Standard Deviation of data stored per node after the increase in cluster size.

5.6 Experiment 3: Temporal Changes

Clusters change over time. Hardware failures and aging machines needs to be replaced. For our third experiment we wanted to look at how a couple mutations to the cluster play out. In this experiment we focused on CRUSH and CRUSH'.

Figure 5.2 illustrates how the cluster changes over time. We simulate a 50% increase (10 to 15 nodes), then the loss of a single node (15 to 14 nodes), followed by another 40% increase (14 to 20 nodes). These systems are designed to be usable during migrations and failures. Therefore after each migration we add more files to the cluster. Table 5.3 shows how many files and how much data is stored in the cluster after each migration.

We used CRUSH' and CRUSH to distribute 1024 buckets and the files where stored as eight blocks, three of which were generated by erasure coding. The results are available in

	CRUSH'	CRUSH	Hashchain'	Hashchain	Consistent'	Consistent	Ring	Shuffle'
4%	0.68	20.46	0.8	0.71	1.06	2.43	0.86	11.17
8%	1.24	19.97	1.11	0.97	1.50	2.8	1.05	14.26
12%	1.22	1.1	1.04	1.18	1.96	3.09	1.24	13.34
16%	1.2	18.09	1.4	1.32	1.98	3.56	1.37	15.39
20%	1.46	1.4	1.3	1.39	2.38	3.83	1.38	13.92
24%	1.52	1.37	1.48	1.56	2.75	3.96	1.65	14.62
28%	1.59	1.38	1.48	1.74	2.78	4.11	1.69	11.85
32%	1.73	1.62	1.29	1.72	3.00	4.03	1.97	12.82
36%	1.81	1.67	1.65	1.82	3.07	4.16	1.86	13.94
40%	1.85	1.8	1.62	1.91	3.12	4.05	1.96	12
44%	1.83	1.73	1.46	2.04	3.16	4.02	1.56	12.67
48%	1.71	1.88	1.53	1.83	3.70	3.9	1.58	12.03
52%	2	1.9	1.88	1.78	3.68	3.82	1.33	11.71
56%	2	2.09	1.78	1.92	3.53	3.77	1.46	11.06
60%	2.1	1.83	1.54	1.94	3.62	3.44	1.68	11.28
64%	2.15	1.96	1.83	1.9	3.59	3.3	1.96	10.73
68%	2.09	1.83	1.74	2.1	3.70	3.53	1.67	10.32
72%	2.05	1.93	1.86	2.14	3.70	3.65	2.16	9.99
76%	2.19	1.88	1.99	2.19	3.77	3.57	2.22	9.66
80%	2.2	1.98	1.99	2.2	3.83	3.61	2.35	9.33
84%	2.2	2.01	1.86	2.17	3.80	3.51	2.1	9.47
88%	2.05	2.09	1.77	2	3.60	3.51	2.02	8.76
92%	2.15	2.1	1.95	2.15	3.59	3.48	2.1	8.59
96%	2.17	2.02	1.87	1.94	3.75	3.44	2.3	8.41
100%	2.11	1.94	1.77	2.02	3.65	3.33	2.64	8.03

Table 5.2: Experiment 2 Scaling: The standard deviation, in GiB, after increasing the cluster by x%, distributing 2.8 TiB of data. With regards to a 100% node increase, Ring moved the least amount but it was middle of the pack in evenly distributing that data. To our surprise when adding 1, 2, or 4 nodes CRUSH had the worst distribution of data, we tested this with a greater number of input files but the result remained.

Node Count	10	15	14	20
Files	225,000	300,000	1,000,000	1,000,000
Bytes (TiB)	2.86	3.81	12.71	12.71

Table 5.3: Experiment 3 Sizes: During the experiment not only did we add and remove nodes, we also stored more and more data to the system to better emulate production environments. To start with we stored a quarter million files on ten nodes. After adding five more nodes we stored an additional 75,000 files. Finally before the migration from 14 to 20 nodes the cluster stored one million files at over 12 TiB including the erasure coding overhead.

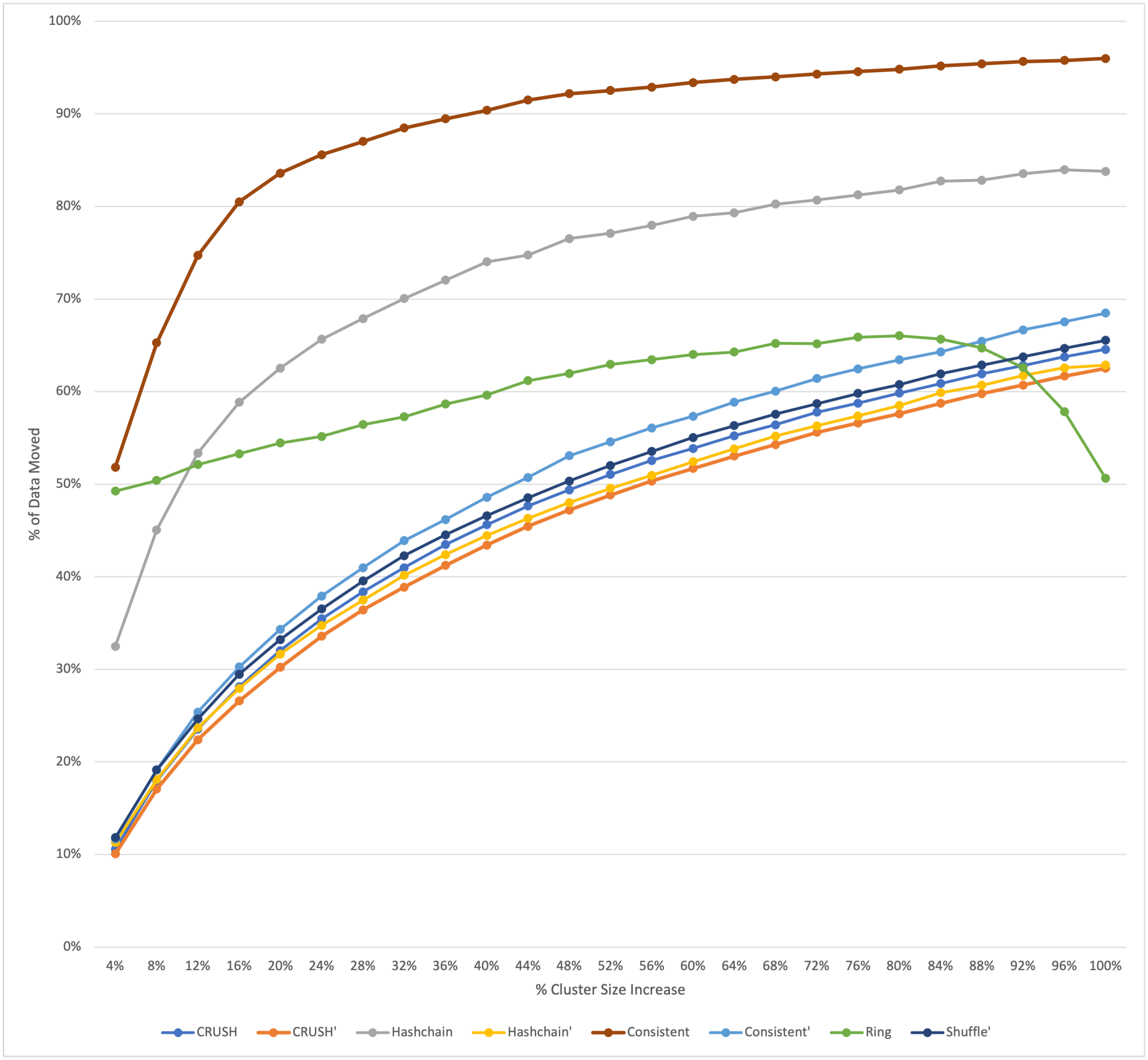

Figure 5.1: Experiment 2 Scaling: Comparison of percentage of data moved by each method moved as the cluster increases by x%. We used erasure coding parameters 20 and 5, split files into 20 blocks and generate 5 additional parity blocks. We can see Consistent Hashing with virtual hosts (100 hosts per node) and Hashchain perform the most unnecessary moves. When only increasing the cluster by a few nodes ($< 8\%$) most of the methods perform as well as CRUSH. CRUSH' provides minimal data movement until the cluster nearly doubles in size. Unexpectedly the ring method performs much better when doubling a cluster in size, however at smaller increases it performs much worse, and doubling ones cluster is not a viable long term solution.

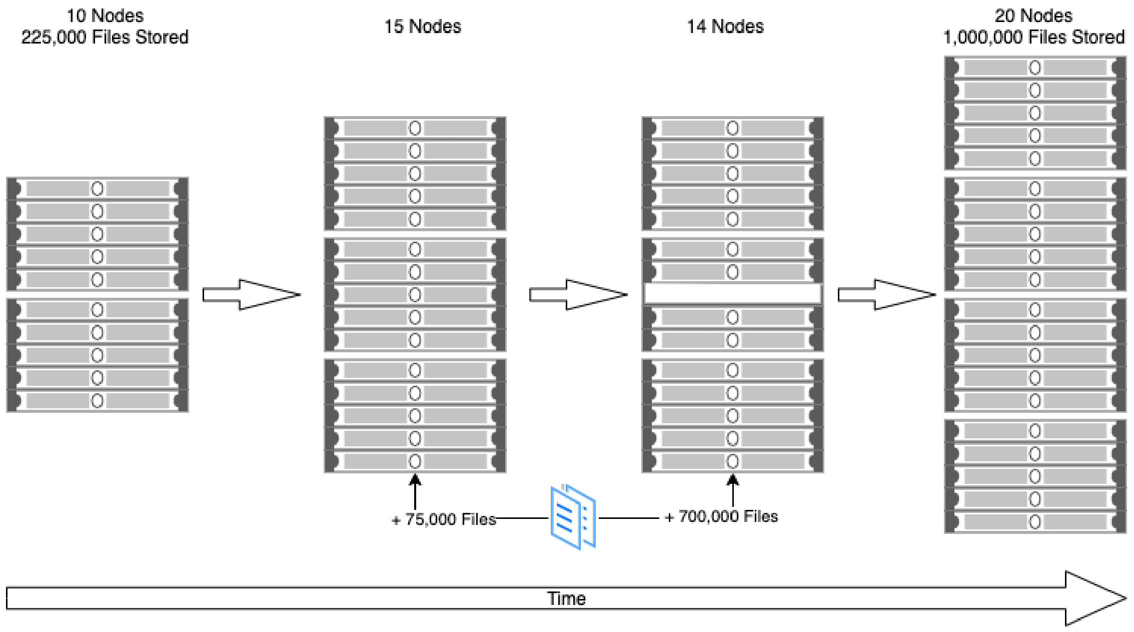

Figure 5.2: Experiment 3 Temporal Changes: compares our CRUSH' with CRUSH over time. We start with 10 storage nodes and add 5 more. After a short period a node is lost due to hardware failures and a replacement must be ordered. As the one node is replaced the organization ordered an additional 5 nodes bringing the cluster up to 20 nodes. Additional files are added at each stage. Over the course of the experiment the number of files stored goes from 225,000 files to 1,000,000.

	Add 5 Nodes		Lose 1 Node		Add 6 Nodes	
	CRUSH'	CRUSH	CRUSH'	CRUSH	CRUSH'	CRUSH
Blocks Moved	765,818	806,996	205,586	236,447	2,892,622	3,102,905
Bytes Moved (GiB)	1,247.23	1,308.58	331.42	385.13	4,741.33	5,069.75
Std Dev (GiB)	4.9	7.44	8.43	9.3	24.68	27.22

Table 5.4: Experiment 3 Data: As we can see in each migration CRUSH' moved less information in both total number of blocks and raw bytes. At the end of the experiment, the cluster stored 12.7 TiB and CRUSH moved an additional .44 TiB over CRUSH'. As the amount of data stored in the system increases so does the standard deviation, in all migrations the resulting distribution of CRUSH' has a smaller standard deviation than CRUSH.

Figure 5.4. In all three migrations CRUSH' moved less data and had a smaller standard deviation than CRUSH.

Chapter 6

Conclusions

Distributing parity information on unique nodes such that changes in a cluster lead to efficient migrations is difficult. How an algorithm handles collisions greatly influences the amount of data that must be migrated. When a collision is encountered a dependency between the colliding blocks and all remaining blocks is made. Naive approaches like hashchain A.1.1 exacerbate this dependency. In Section 6.1 we go over the set of rules we found improve all these naive attempts. Finally in Section 6.2 we discuss future work.

6.1 Rules

When searching for ways to reduce data movement due to scaling in our hashchain distribution method A.1.1 we discovered several modifications which improved all of the methods we looked at. Our research has shown that placing all non-colliding blocks each pass and augmenting the key for every block has the potential to greatly improve basic distribution schemes. Furthermore we noticed that some methods performed best by removing nodes as potential candidates , while others were improved by augmenting the key by encoding how many passes the method has done to resolve collisions.

6.1.1 Rule: Defer Handling Collisions

When trying to create a list of n unique hosts to place the blocks of an object defer placing blocks that collide to future passes. By only placing blocks that do not collide we saw a dramatic reduction in the amount of data moved. This idea is built into CRUSH [28] and when added to Hashchain A.1.1, Consistent A.1.4, and Shuffle A.1.6 we saw a large reduction in the amount of data movement during migrations.

6.1.2 Rule: Place Blocks Independently

Compute the placement of each block separately. Consistent A.1.4 moved a lot of data simply because it placed all the blocks in relation to the first one. If a new node got inserted between the hashes of the key and original node then all the blocks will need to move as well. By adding multiple passes, see 6.1.1, and placing blocks based off of the hash of the key and block index Consistent' A.1.5 was able to move significantly less data without affecting how even the data was spread. Our Hashchain' A.1.2 also improved using this technique. By spinning off new chains from colliding hashes, the result was block placements became more independent.

6.1.3 Rule: Encode How Many Passes in Keys for Blocks

We also noticed that in some cases, removing the used nodes before resolving collisions reduced the amount of data that must be migrated. Our CRUSH' 4.2 derivative removes all the nodes used at the end of each pass. This minor change provided around a 2% reduction in data moved during migrations. In other cases instead of removing nodes it was better to augment the key for placing a block by encoding how many passes had taken place. In Hashchain' A.1.2 each pass tried hashing the hash generated previously.

6.1.4 Applying the Rules

The naive Shuffle method, using a single array shuffle and selecting the first n nodes, is a prime example of how applying these rules can make a method viable.

In one of our tests Shuffle moved over 98% of all the stored data, including the parity information. We applied the new rules and created Shuffle' A.1.6. First every single block index was used to perform a shuffle of the array. Each round the node at the first index was considered, if placing the block on this node would cause a collision, we moved on to the next block index. After each round we would check if another pass was needed. The key for each block index was the concatenation of the key, block index, and pass count. The result was an algorithm that only moved 61% of the data which was just greater than CRUSH and Hashchain'.

6.2 Future Work

This research merely opens the door to many other questions.

- Understanding why the rules outlined in 6.1 reduced the amount of data moved.
- If we understand why, are there other techniques that would improve migration further
- Further understanding of how these changes affect the distribution of data. Shuffle A.1.6, when compared to the naive approach, greatly reduced the amount of data moved in migrations but at the cost of increasing the variance in node usage. Could including nodes multiple times like Dynamo's Consistent Hashing make shuffle move data more efficiently and keep the standard deviation small?
- Experiment 2 Scaling 5.5 showed that the naive ring approach performed the best when doubling the size of a cluster. While its not practical to only grow or shrink one's storage by a factor of two, with further understanding this effect might be usable to improve Consistent Hashing.
- During our experiments we noted that both CRUSH and CRUSH' could produce uneven distributions compared to other methods. This is highlighted in Experiment 2 Scaling 5.5 where increasing the number of nodes by 4% lead to CRUSH providing the largest standard deviation. Understanding why and what types of migrations are best avoided would help alleviate headaches of system administrators with full clusters.
- We barely touched Consistent Hashing 3.5. Our implementation of Dyanmo's approach [9], Consistent A.1.4, and our improved version, Consistent' A.1.5, leaves out a lot of research in this area. Cassandra [15], while based on Dynamo, does not duplicate nodes around the circle, ECHash [7] uses different circles for each block, and ZHT [16]

defines equal sized partitions and a maximum number of nodes to help balance the data distribution.

- All our research is based on the assumption that every node is of equal capacity, resources, and no failures. While we hope our rules 6.1 hold up in heterogeneous clusters and with failures these cases still must be tested.
- Our research focuses on distributing a file to a set of nodes. CRUSH and CRUSH' work on complex hierarchies to describe an organization's infrastructure. One can easily define rules to not only keep data on separate nodes but also separate server racks. CRUSH' has not been tested against these types of layouts.

Bibliography

[1] 45Drives. What is ceph why our customers love it, 1 2019. URL: `http://45drives.blogspot.com/2019/01/what-is-ceph-why-our-customers-love-it.html`.

[2] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[3] SB Balaji, M Nikhil Krishnan, Myna Vajha, Vinayak Ramkumar, Birenjith Sasidharan, and P Vijay Kumar. Erasure coding for distributed storage: An overview. *Science China Information Sciences*, 61(10):1–45, 2018.

[4] Hari Balakrishnan, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, 2003.

[5] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[6] Josh Cates. *Robust and efficient data management for a distributed hash table.* PhD thesis, Massachusetts Institute of Technology, 2003.

[7] Liangfeng Cheng, Yuchong Hu, and Patrick PC Lee. Coupling decentralized key-value stores with erasure coding. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 377–389, 2019.

[8] Frank Frank Edward Dabek. *A distributed hash table.* PhD thesis, Massachusetts Institute of Technology, 2005.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[10] Gluster. Glusterfs algorithms: Distribution, 3 2012. URL: `https://www.gluster.org/glusterfs-algorithms-distribution/`.

[11] R.J. Honicky and E.L. Miller. Replication under scalable hashing: a family of algorithms for scalable decentralized data distribution. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 96–, 2004. https://doi.org/10.1109/IPDPS.2004.1303042 `doi:10.1109/IPDPS.2004.1303042`.

[12] @jerryxu. Libcrunch and crush maps, 6 2013. URL: `https://blog.twitter.com/engineering/en_us/a/2013/libcrunch-and-crush-maps`.

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

[14] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11-16):1203–1213, 1999.

[15] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[16] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 775–787, 2013. https://doi.org/10.1109/IPDPS.2013.110 `doi:10.1109/IPDPS.2013.110`.

[17] Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In *International Workshop on Peer-to-Peer Systems*, pages 88–97. Springer, 2003.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[19] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[20] Garret Swart. Spreading the load using consistent hashing: A preliminary report. In *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 169–176. IEEE, 2004.

[21] A Tada, Kenichi Hirosawa, Fumihiko Kannari, M Takeoka, and M Sasaki. Photon-number squeezing in a soliton-like stokes raman component split during fiber propagation. In *International Quantum Electronics Conference, 2005.*, pages 1014–1015. IEEE Computer Society, 2005.

[22] Daniel Van der Ster and Arne Wiebalck. Building an organic block storage service at cern with ceph. In *Journal of Physics: Conference Series*, volume 513, page 042047. IOP Publishing, 2014.

[23] Alexey Vanin, Vladimir Bogatyrev, and Stanislav Bogatyrev. Data migration rate of crush-based distributed object storage with dynamic topology. In *International Conference on Distributed Computer and Communication Networks*, pages 464–471. Springer, 2020.

[24] Xiaoming Wang and Dmitri Loguinov. Load-balancing performance of consistent hashing: asymptotic analysis of random node join. *IEEE/ACM Transactions on Networking*, 15(4):892–905, 2007.

[25] Sage A Weil. crush: straw is dead, long live straw2, 2014. URL: `https://www.spinics.net/lists/ceph-devel/msg21635.html`.

[26] Sage A Weil. libcrush is a c library to control placement in a hierarchy, 2017. URL: `https://github.com/ceph/libcrush`.

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

[28] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31, 2006. https://doi.org/10.1109/SC.2006.19 `doi:10.1109/SC.2006.19`.

[29] Jiang Zhou, Yong Chen, and Weiping Wang. Atributed consistent hashing for heterogeneous storage systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–12, 2018.

[30] Jiang Zhou, Wei Xie, Qiang Gu, and Yong Chen. Hierarchical consistent hashing for heterogeneous object-based storage. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1597–1604. IEEE, 2016.

Appendix A

Distribution Strategies

There are many ways to distribute files across a cluster. One could assign hash ranges to nodes in the cluster and then match filename hashes to the owning node. Another could arrange their nodes in a ring, each with an index from $0..N$, then $HASH(name)$ **mod** N could be used to identify the node. Algorithms like straw use a deterministic approach akin to drawing straws [28]. While these methods are straight forward, using them to distribute an erasured file while limiting the number of pieces that can coexist on a node can lead to inefficient data migration later on.

A.1 Methods

In this section we will give an overview of all the different methods we tested to distribute data across a cluster.

A.1.1 Hashchain

Hashchain was our initial naive attempt at distributing files on our cluster. Like other DHT systems we used the hash of the filename to create a sequence of hashes dictating where each block of a file should be stored. For example, when distributing a file broken into three separate blocks hashchain would produce a sequence like so $HASH(filename)$, $HASH(HASH(filename))$, $HASH(HASH(HASH(filename)))$. To uphold the one block per node policy we ignorantly skipped hashes that cause collisions. Let's assume in our previous example the second block's hash is a collision with the first, therefore the sequence would become [$HASH(filename)$, $HASH(HASH(HASH(filename)))$, $HASH(HASH(HASH(HASH(filen$]. At the time we didn't consider the ramifications of this decision, and in hindsight it is painfully clear how we coupled the position of subsequent blocks with previous ones causing cascades of movement when hashes get assigned to other nodes.

Algorithm 3 Hashchain Method: Our initial attempt at distributing files across a cluster. Its poor performance when resizing a cluster is what led us to researching alterations and alternatives.

```
$HASH2NODE \leftarrow$ retrieves what node a hash belongs to
$next \leftarrow HASH(filename)$
$chain \leftarrow [HOST2NODE(next)]$
while $|chain| < n$ do
    $next \leftarrow HASH(next)$
    $node \leftarrow HOST2NODE(next)$
    if $nodenotinchain$ then
        $APPEND(node)$
    end if
end while
return $chain$
```

A.1.2 Hashchain'

We built a simulator to figure out why our Hashchain method was performing so poorly on cluster resize and to see if we could improve it. We made many modifications to the algorithm and began studying other methods out there. Hashchain' is the product of several iterations to the Hashchain method. Through our simulator we had an easy way to distribute hundreds of thousands of files quickly and generate statistics about the evenness of data being distributed and how much data was moved given a cluster layout change.

Hashchain' works by no longer skipping hashes. Each block's hash is derived from the previous block's hash as before: $HASH(filename)$, $HASH(HASH(filename))$, ..., $HASH(HASH(...HASH(filename)))$. To handle collisions we alter the block's hash by increment it then creating a new hash chain from that. For example if block 1 collides with block 0 the next time we try to assign block 1 we use $HASH(HASH(HASH(filename) + 1))$. This alteration greatly reduces the number of subsequent blocks that must also move when a cluster change removes a previous collision.

Note while the algorithm A.1.2 uses filename to distribute files, bucket ids can also be used which is what we do in our experiments section.

Algorithm 4 Hashchain' Method: After several iterations this alteration to our Hashchain method provides similar performance to CRUSH. On collision we spawn new hash chains for each collision by adding one to the root hash.

```
$HASH2NODE \leftarrow$ retrieves what node a hash belongs to
$chain \leftarrow [HASH(filename)]$
$seen \leftarrow \{HASH2NODE(chain[1])\}$
$revisit \leftarrow \{\}$
for $i \leftarrow 2$ upto $n$ do
    $chain[i] \leftarrow HASH(chain[i-1])$
    $node \leftarrow HASH2NODE(chain[i])$
    if $node \notin seen$ then
        $seen \leftarrow seen \cup node$
    else
        $revisit \leftarrow revisit \cup \{i\}$
        $chain[i] \leftarrow chain[i] + 1$
    end if
end for
while $|revisit| > 0$ do
    for $i \leftarrow 2$ upto $n$ do
        if $i \in revisit$ then
            $chain[i] \leftarrow HASH(chain[i])$
            $node \leftarrow HASH2NODE(chain[i])$
            if $node \notin seen$ then
                $seen \leftarrow seen \cup \{node\}$
                $revisit \leftarrow revisit - \{i\}$
            end if
        end if
    end for
end while
return $chain$
```

A.1.3 Ring

Arranging all the nodes in a cluster as a circle is a very popular method for distributing blocks of a file. Like standard Distributed Hash Tables there is only a single hash to calculate, all blocks after the first are on the next node in a defined order. Amazon's Dynamo [9] is a primary example of using this strategy. As we say in 5 it does a decent job in moving data on cluster changes however it lags behind other methods except when doubling a cluster in size. Systems like Dynamo will go a step further and put spacing between nodes to avoid hot spots when nodes are added and removed. We did not replicate this behavior but stuck to the basic ring strategy A.1.3.

Algorithm 5 The Ring Method uses a hash of the filename to determine the first node to place the file, then iterates through nodes in a predefined order till all blocks are placed. A simple AFTER function would sort the hosts, find the nodes index, then add one and get the modulo from number of nodes.

```
$HASH2NODE \leftarrow$ retrieves what node a hash belongs to
$next \leftarrow HOST2NODE(HASH(filename))$
$chain \leftarrow [next]$
**for** $i = 2$; $i \leq n$; $i += 1$ **do**
    $next \leftarrow AFTER(next)$
    $APPEND$(chain, $next$)
**end for**
**return** $chain$
```

A.1.4 Consistent

One of drawbacks of Ring A.1.3 in distributed clusters is the dependent relationship between number of nodes and where buckets are stored. By using $HASH(bucketid) mod |nodes|$ the simple addition or subtraction of nodes change the resulting index for lots of buckets. To address this issue Consistent Hashing [13] was introduced. In simple terms instead of assigning each node a range of hashes it is responsible we hash an identifier of a node to a spot on a circle, once all nodes are placed around the circle we can hash the key of an object to store and that key will be placed on the next node that hashes to a value $\geq$ to the key's hash. If the key's hash is ¿ than all node hashes it wraps around the the first server.

A side effect of using a hash to place nodes in the circle is one cannot guarantee each node will have an equal share of the circle assigned to it. This can lead to the standard deviation of data stored being quite high. Another effect is if a node is lost the new node to receive the data will be overloaded since all recovery will happen in a specific part of the circle. To address these issues one can include nodes multiple times in the circle, DynamoDB [9] calls these virtual hosts. Mapping nodes multiple times reduces the variance between how much of the circle each node is responsible for and it spreads the load on recovery when a node is removed from the circle since a node is no longer followed by a single node. DynamoDB also uses virtual hosts as a way to handle heterogeneous capacities, smaller nodes have fewer placements in the circle.

The pseudo code for Consistent Hashing is available at A.1.4. It is a basic implementation that uses the hash of the key and the block index to determine where on the circle to start. Any node entry that points to an existing node will simply be skipped. We did try a derivative where each block index had its own circle, nodes were placed using the block index in the hash, however this performed worse than the basic strategy outlined above. Other alterations exist but have not been tested in our simulator. ECHash [7] for example attempts to specialize the approach for storing erasure coding data using multiple circles, called hash rings. Another project ZHT [16] places places blocks both clockwise and counterclockwise around the key's hash.

A.1.5 Consistent'

Just like for CRUSH' and Hashchain' we can apply our rules to the basic consistent hashing algorithm to reduce how much data is moved. We found that by placing each block by hashing the key with the block id and then finding the first unused node by walking around the ring greatly reduced the cost. Algorithm A.1.5 shows these changes.

A.1.6 Shuffle

The simulator made it very easy to test different ideas. A silly one to consider is a simple shuffle routine. Using deterministic pseudo random shuffle we could distribute files by placing all the hosts in an array, sorting, then shuffling in a way that uses the hash of the filename for randomness. This idea took several iterations. We tried setting the random number generator seed to the hash value, with poor results. Also just shuffling the array once

Algorithm 6 Consistent Hashing: This basic algorithm maps both nodes and keys to a circle and associates keys with the next node that is greater than or equal to the key's hash. Generates an n length list of nodes to store all the blocks mapped to the key. Nodes are placed around the ring m times to minimize hot spots on failures and more evenly distribute the data between the nodes. In testing nodes had to be placed around the circle over 100 times to get close to the standard deviation of other methods.

```
$ring \leftarrow []$
**for** $node$ **in** $nodes$ **do**
    **for** $i \leftarrow 1$ **upto** $m$ **do**
        $APPEND(\text{ring}, (HASH(node + i), node))$
    **end for**
**end for**
$SORT(ring, func(a, b)\{return\ \text{a}[0] < b[0]\})$
$index \leftarrow BINARYSEARCH(ring, HASH(key))$
$node \leftarrow ring[index][1]$
$chain \leftarrow [node]$
$used \leftarrow \{node\}$
**for** $i \leftarrow 2$ **upto** $n$ **do**
    **do**
        $index \leftarrow index + 1$ **mod** $|ring|$
    **while** $ring[index][1] \in used$
    $node \leftarrow ring[index][1]$
    $APPEND(\text{chain}, node)$
    $used \leftarrow used \cup \{node\}$
**end for**
**return** $chain$
```

Algorithm 7 Consistent': Derivative of the basic Consistent Hashing algorithm however each block is placed off of a different hash instead of walking around the ring. We found that this simple change greatly reduced the amount of data moved when using the placement algorithm described by DynamoDB.

```
$ring \leftarrow []$
**for** $node$ **in** $nodes$ **do**
    **for** $i \leftarrow 1$ **upto** $m$ **do**
        $APPEND$(ring, $[HASH(node + i), node]$)
    **end for**
**end for**
$SORT(ring, func(a, b)\{return$ a$[0] < b[0]\})$
$chain \leftarrow []$
$used \leftarrow \{\}$
**for** $i \leftarrow 1$ **upto** $n$ **do**
    $hash \leftarrow HASH(key + i)$
    $index \leftarrow BINARYSEARCH(ring, hash)$
    **while** $ring[index][1] \in used$ **do**
        $index \leftarrow index + 1$ **mod** $|ring|$
    **end while**
    $node \leftarrow ring[index][1]$
    $APPEND$(chain, $node$)
    $used \leftarrow used \cup \{node\}$
**end for**
**return** $chain$
```

and taking the first n items moved an exorbitant amount of data. We found that generating the hash with the filename and block index, shuffling a sorted array, and selecting the head host was enough to give comparable results to other methods. A downside to this approach would be trying to apply a weight to certain nodes which have larger capacities over smaller nodes, perhaps larger nodes could be duplicated in the array.

Our shuffle routine was derived from the Fisher-Yates shuffle algorithm

Algorithm 8 The Shuffle Method uses the hash to randomly shuffle an ordered array of hosts. While there are many drawbacks it was interesting to see that it was able to efficiently move blocks like CRUSH and Hashchain'. However the nodes were the most uneven compared to all other methods.

```
procedure SHUFFLE($array$, $seed$)
    for $i \leftarrow |array|$ downto 2 do
        $j \leftarrow FLOOR((HASH(seed, i) \textbf{ mod } 65536) \div 65535) \times i) + 1$
        $tmp \leftarrow array[i]$
        $array[i] \leftarrow array[j]$
        $array[j] \leftarrow tmp$
    end for
end procedure
$hosts \leftarrow$ array of all hosts sorted
$used \leftarrow \{\}$
$revisit \leftarrow \{1..n\}$
$chain \leftarrow []$
while $|revisit| > 0$ do
    for $i \leftarrow 1$ to $n$ do
        if $i \in revisit$ then
            $arr \leftarrow CLONE(hosts)$
            $SHUFFLE(arr, CONCAT(filename, i))$
            $host \leftarrow arr[1]$
            if $host \notin used$ then
                $chain[i] \leftarrow host$
                $used \leftarrow used \cup \{host\}$
                $revisit \leftarrow revisit - \{i\}$
            end if
        end if
    end for
end while
return $chain$
```

Appendix B

libcrush: Patchfile

For our experimentation we used libcrush available at *https* : //*github.com*/*ceph*/*libcrush*/*tree*/*de2e85*
Below is the patch file generated from our modifications to their choose independent algorithm, used when distributing Ceph Placement Groups to store data using erasure coding.

```
diff --git a/crush/mapper.c b/crush/mapper.c
index c71b6140..56587f12 100644
--- a/crush/mapper.c
+++ b/crush/mapper.c
@@ -631,6 +631,30 @@ reject:
  return outpos;
 }


+#include "builder.h"
+static struct crush_bucket *crush_bucket_clone(const struct crush_bucket *bucket)
+{
+ __u32 i;
+ switch (bucket->alg) {
+ case CRUSH_BUCKET_STRAW2: {
+ struct crush_bucket_straw2 *b = malloc(sizeof(*b));
+ *b = *((const struct crush_bucket_straw2 *) bucket);
+ b->h.items = NULL;
+ b->item_weights = NULL;
+ if (b->h.size) {
+ b->h.items = malloc(sizeof(*b->h.items) * b->h.size);
+ b->item_weights = malloc(sizeof(*b->h.items) * b->h.size);
+ memcpy(b->h.items, bucket->items, sizeof(*bucket->items) * bucket->size);
+ memcpy(b->item_weights,
+ ((const struct crush_bucket_straw2 *) bucket)->item_weights,
+ sizeof(*b->item_weights) * bucket->size);
+ }
+ return &b->h;
+ }
+ default:
+ return NULL;
+ }
+}


 /**
  * crush_choose_indep: alternative breadth-first positionally stable mapping
@@ -649,7 +673,7 @@ static void crush_choose_indep(const struct crush_map *map,
         int parent_r,
                                    const struct crush_choose_arg *choose_args)
 {
- const struct crush_bucket *in = bucket;
+ struct crush_bucket *in;
  int endpos = outpos + left;
  int rep;
  unsigned int ftotal;
@@ -662,6 +686,11 @@ static void crush_choose_indep(const struct crush_map *map,
  dprintk("CHOOSE%s INDEP bucket %d x %d outpos %d numrep %d\n", recurse_to_leaf ? "_LEAF" : "",
  bucket->id, x, outpos, numrep);


+ /* after each pass we append all the ids that were appended to swipe_used. */
+ int used_count = 0;
+ int used[256];
+ if (numrep > 256) abort();
+
  /* initially my result is undefined */
  for (rep = outpos; rep < endpos; rep++) {
  out[rep] = CRUSH_ITEM_UNDEF;
@@ -669,6 +698,9 @@ static void crush_choose_indep(const struct crush_map *map,
  out2[rep] = CRUSH_ITEM_UNDEF;
  }


+ /* after each swipe/pass we remove items that have been used */
+ in = crush_bucket_clone(bucket);
+
  for (ftotal = 0; left > 0 && ftotal < tries; ftotal++) {
 #ifdef DEBUG_INDEP
  if (out2 && ftotal) {
@@ -684,12 +716,21 @@ static void crush_choose_indep(const struct crush_map *map,
  dprintk("\n");
  }
 #endif
+
+ dprintk("item_count: %u\n", in->size);
+ int u;
+ for (u = 0; u < used_count; ++u) {
+ dprintk("removing item %u from bucket\n", used[u]);
+ crush_bucket_remove_item(NULL, in, used[u]);
+ }
+ used_count = 0;
+ dprintk("item_count: %u\n", in->size);
+
+
  for (rep = outpos; rep < endpos; rep++) {
  if (out[rep] != CRUSH_ITEM_UNDEF)
  continue;


- in = bucket;  /* initial bucket */
-
  /* choose through intervening buckets */
  for (;;) {
  /* note: we base the choice on the position
```

```
@@ -700,16 +741,15 @@ static void crush_choose_indep(const struct crush_map *map,
   * this will involve more devices in data
   * movement and tend to distribute the load.
   */
- r = rep + parent_r;
+ // r = rep + parent_r;
+ r = rep;


- /* be careful */
- if (in->alg == CRUSH_BUCKET_UNIFORM &&
-      in->size % numrep == 0)
- /* r'=r+(n+1)*f_total */
- r += (numrep+1) * ftotal;
- else
- /* r' = r + n*f_total */
- r += numrep * ftotal;
+ /*
+ * crush skips ahead n spaces to place blocks that collide or would go to
+ * down/out of map items. Instead by removing those items each swipe we
+ * provide the same guarantee that a disk coming back online doesn't
+ * cause all other blocks following to need adjustment as well.
+ */


  /* bucket choose */
  if (in->size == 0) {
@@ -794,6 +834,7 @@ static void crush_choose_indep(const struct crush_map *map,
  /* yay! */
  out[rep] = item;
  left--;
+ used[used_count++] = item;
  break;
  }
  }
@@ -824,6 +865,7 @@ static void crush_choose_indep(const struct crush_map *map,
  dprintk("\n");
  }
 #endif
+ crush_destroy_bucket(in);
 }
```

Appendix C
Source Code

The source code for our simulator is provided below. One can run a test with bash like so $catfiles.txt \mid ./placement.pl < (./gentest.pl25301620)hashchain{-}prime1024 \mid ./analyze.pl$. This command will read all the files and their sizes from $files.txt$ and distribute them on a cluster of 25 nodes, files will be split into 16 equal parts and 4 erasure blocks will be added, meaning each file is stored in 20 blocks total. Finally we use the $hashchain - prime$ to distribute 1024 buckets.

We provide the following files below. The *gentest.py* script C.1 is a Python 3 script which builds the before and after clusters, evenly distributing the available hashes for the Hashchain and Hashchain' algorithms. The two primary scriops *placement.pl* C.2 and *crush* application C.3 output the size of each block and how that block will be migrated between the two clusters. The *analyze.pl* C.4 script reads the output of either *placement.pl* or *ceph.cpp* and creates a report with basic statistics. Finally *common.pl* C.5 holds the implementations of all the methods handled by *placement.pl*.

C.1 gentest.py

This script was made to make building tests easier. It evenly divides a 32bit hash space between M nodes, then resizes the cluster to N nodes by reassigning hashes to the new nodes. It was used with Python 3.8 and can be run like $./gentest.py25301620$ where 25 is starting number of nodes in the cluster, 30 is the number of nodes to migrate to, 16 is how many blocks we will split files into, and 20 is the number of blocks each file will be stored with in the cluster, including erasure coding data.

```
#!/usr/bin/env python3
```

```
# Copyright 2019 United States Government as represented by the Administrator
# of the National Aeronautics and Space Administration.  All Rights Reserved.
import copy, random, sys, uuid

def dump(varname, m):
    print(f"our %{varname} = (")
    for j, nb in enumerate(m.items()):
        node, buckets = nb
        #"node-000",[[0x00000000,0x08d3dcb0]],
        print(f'  "{node}"', end=",[")
        for i, b in enumerate(buckets):
            if i != 0:
                print(",", end="")
            print("[0x%08x,0x%08x]" % b, end="")
        if (j + 1) < len(m):
            print("],")
        else:
            print("]")
    print(");")

def bucketsize(b):
    return (b[1] - b[0]) + 1

def dumpsizes(m):
    total = 0
    for node, buckets in m.items():
        sum = 0
        for b in buckets:
            sum += bucketsize(b)
        print(f'{node}: {sum}')
        total += sum
    print("total:", total)

def validate(m):
    """ make sure that all buckets exist in the map """
    find = 0
    while (find + 1) < (1 << 32):
        found = False
        for buckets in m.values():
            for b in buckets:
                if b[0] == find:
                    found = True
                    find = b[1] + 1
        if not found:
            raise Exception("missing bucket 0x%08x" % find)
    # make sure we have the last value
    for bucket in m.values():
        for b in buckets:
            if b[1] == 0xffffffff:
                return True
    raise Exception("missing bucket 0xffffffff")

def dumpweights(numnodes):
    # set the seed to make repeatable
    gen = random.Random(111)
    print("our %HOSTS = (")
    for i in range(1, numnodes + 1):
        raw = gen.getrandbits(8 * 16)
        if i > 1:
            print(",\n", end="")
        print('  "node-%03d"' % i, ',["', hex(raw)[2:], '",1.0]', sep="", end="")
    print("\n);")

def setup(n):
    """ create a dict of all the hostscalculate buckets per node and the remainder given n nodes """
    nodes = { "node-%03d" % host: list() for host in range(1, n + 1) }
    bpn = int((1 << 32) / n)
    remainder = (1 << 32) % n
    bookmark = 0
    for k, v in nodes.items():
        # take 1 off of bpn as we are 0 based and the ranges are inclusive
        step = bpn - 1
        if remainder > 0:
            step += 1
            remainder -= 1
        v.append((bookmark, bookmark + step))
        bookmark += step + 1
    return nodes

def addto(before, n):
    """ make the map n nodes """
    nodes = copy.deepcopy(before)
    bpn = int((1 << 32) / n)
    remainder = (1 << 32) % n
    new = set()
    for i in range(len(nodes) + 1, n + 1):
        node = "node-%03d" % i
        new.add(node)
        nodes[node] = list()

    # the set of bucket ranges available for us to give to the new nodes
    pool = list()
    for node, buckets in nodes.items():
        if node not in new:
            # shrink existing buckets
            # ASSUMES only a single bucket per node
            step = bpn - 1
            if remainder > 0:
                step += 1
                remainder -= 1
            start, end = buckets[0]
            pool.append((start + step + 1, end))
            buckets[0] = (start, start + step)

    for node, buckets in nodes.items():
        if node in new:
            count = bpn
            if remainder > 0:
                count += 1
                remainder -= 1

            while count > 0:
                next = pool.pop(0)
                nextsize = bucketsize(next)
                # print(count, "vs", nextsize)
                if nextsize == count:
                    # print("==")
                    buckets.append(next)
                    count = 0
                elif nextsize > count:
                    # print("<")
                    # print("reducing %08x,%08x to %08x,%08x" %(next[0], next[1], next[0] + count + 1,
                         next[1]))
                    buckets.append((next[0], next[0] + (count - 1)))
                    pool.insert(0, (next[0] + count, next[1]))
                    # print(bucketsize((next[0] + count, next[1])), "-------------", count)
                    count = 0
                else:
                    # print(">")
                    buckets.append(next)
                    count -= nextsize
    return nodes

args = sys.argv[1:]
if len(args) != 4:
    print("Usage: genmap.py FROM TO K M")
    print("   Ex: genmap.py 25 26 20 25")
    sys.exit(1)

before = setup(int(args[0]))
try:
    validate(before)
except Exception as e:
    print("invalid before map: ", e)
    dump("before", before)
    dumpsizes(after)
    sys.exit(1)

after = addto(before, int(args[1]))
try:
    validate(after)
except Exception as e:
    print("invalid after map: ", e)
    dump("after", after)
    dumpsizes(after)
    sys.exit(1)

print(f"our $K = {args[2]};")
print(f"our $M = {args[3]};")
print("our $ERASURE_THRESHOLD = 0;")
dumpweights(int(args[1]))
dump("FROM", before)
dump("TO", after)
```

C.2 placement.pl

This is the Perl script that calculates where blocks of files should be stored and where they are to be moved. The files are piped into the script, one per line, and file size followed by the path to the file. Only the last component of the path is used to place files, therefore the paths don't have to be exact. To distribute files it uses the number of buckets and method specified, from the command line, to distribute the buckets, then simply hashes and uses modulo to place files in a bucket. This script relies on two other script files, the output of *gentest.py* C.1 and *common.pl* C.5. The former provides the hashing layout for the Hashchain family of methods while the latter holds the implementations of all the distribution methods.

```
#!/usr/bin/env perl
# Copyright 2019 United States Government as represented by the Administrator
# of the National Aeronautics and Space Administration.  All Rights Reserved.
use strict;
use warnings;
use File::Basename;
use Data::Dumper;

# includes all the strategies and hash functions
require "common.pl";

# Usage: pgplacement.pl CLUSTER.pl METHOD PGNUM <files
#
# CLUSTER.pl    perl script that defines the required variables and lays out
#               the cluster and how it's going to change.
# METHOD        name of the distribution method to use
# PGNUM         optional, Enable this many Placement Groups. Default: 0.
# <STDIN>       1 file per line of stdin of format 'SIZE PATH'

# generate a csv/tsv of where all file blocks go and how much space each block
# is taking up.

# defined in common.pl
our %METHODS;
our %HOST_METHODS;
our %HASHES;

$Data::Dumper::Sortkeys = 1;

#------------------------------------------------------------------ Required Vars

# these variables must be defined in the CLUSTER.pl which describes the cluster
# layout we are testing.

# hash of { hostname -> [ BUCKETS, ... ] }
```

```
our %FROM;
our %TO;

our $K;
our $M;

#------------------------------------------------------------------------ BEGIN SCRIPT

print "# " . join(" ", @ARGV) . "\n";

print "# loading $ARGV[0]\n";
require $ARGV[0];

my $method = $ARGV[1];
if (exists $METHODS{$method}) {
    print "# Method: $ARGV[1]\n";
} else {
    print "unknown distribution method: '$ARGV[1]'\n";
    exit 1;
}

our $PGNUM = 0;
if (scalar(@ARGV) > 2) {
    print "# PGNUM: $ARGV[2]\n";
    $PGNUM = $ARGV[2];
}

our $HASH_FUNCTION = \&hash_murmur3;

my %from_index = ();
&build_index(\%from_index, \%FROM);
my %to_index = ();
&build_index(\%to_index, \%TO);

# { pgnum => [ host0, host1, host2, .., hostm ] }
my @from = &distribute_pgs(\%from_index, $PGNUM, $M, $method);
my @to = &distribute_pgs(\%to_index, $PGNUM, $M, $method);

while (<STDIN>) {
    $_ =~ /^(\d+) (.+)$/;
    my $size = int($1);
    my $path = basename $2;
    my $bsize = int($size / $K);
    my $pg = unpack("Q", $HASH_FUNCTION->($path)) % $PGNUM;
    for my $bid (0 .. $M - 1) {
        my $pre = $from[$pg]->[$bid];
        my $post = $to[$pg]->[$bid];
        print "$path,$bid,$pre,$post,$bsize,$M\n";
    }
}
exit 0;

#-------------------------------------------------------------------------- END SCRIPT

sub distribute_pgs {
    my ($dist, $pgnum, $m, $strategy) = @_;
    my @pgs = ();
    for my $i (0 .. $pgnum - 1) {
        # print STDERR "distributing pg $i $m times with $strategy\n";
        my $chain = $METHODS{$strategy}->($dist->{buckets}, $m, "$i");
        # check if the chain has hostnames or hashes in it
        if (exists $HOST_METHODS{$strategy}) {
            push @pgs, $chain;
        } else {
            my @hosts = ();
            for my $link (@{$chain}) {
                push @hosts, &locate($dist->{buckets}, $link);
            }
            push @pgs, \@hosts;
        }
    }
    return @pgs;
}

sub build_index {
    my ($idx, $src) = @_;
    $idx->{type} = "link";
    $idx->{buckets} = {};
    $idx->{pgs} = {};
    for my $hostname (keys %{$src}) {
        for my $bucket (@{$src->{$hostname}}) {
            $idx->{buckets}->{pack("LL", @{$bucket})} = $hostname;
        }
    }
}

# locate which host based off of the bucket mappings
sub locate {
    my ($dist, $link) = @_;
    my $bucket = unpack "L>", $link;
    for my $range (keys %$dist) {
        my ($begin, $end) = unpack("LL", $range);
        return $dist->{$range} if ($bucket >= $begin and $bucket <= $end);
    }
    die "cannot locate host for bucket " . sprintf("%08x", $bucket);
}
```

C.3 crush.cpp

The input and output format to this application is the same as *placement.pl* C.2 however instead of using distribution methods defined in *common.pl* C.5 this application uses an installed version of *libcrush*. Depending on the compilation of the library this application will test placement of input files using $CRUSH$ or $CRUSH'$ if B has been applied to *libcrush*. The other dependency is a simple MurmurHash3 C implementation available on GitHub.

```
/*
 * Copyright 2019 United States Government as represented by the Administrator
 * of the National Aeronautics and Space Administration.  All Rights Reserved.
 */
extern "C"{
// libcrush available at https://github.com/ceph/libcrush
#include <crush/builder.h>
#include <crush/hash.h>
#include <crush/mapper.h>
// murmur3.h and murmur3.c available at https://github.com/PeterScott/murmur3
#include "murmur3.h"
}

#include <cstdio>
#include <cstdlib>
#include <string>
#include <vector>
#include <unordered_map>

// https://ceph.io/geen-categorie/crushmap-example-of-a-hierarchical-cluster-map/

//                              ROOT
//    HOST1        HOST2        HOST3       HOST4       HOST5  ...

enum {
    TYPE_DEFAULT = 0, TYPE_ROOT=1, TYPE_HOST=2
};

struct input {
    std::string name;
    size_t size;
    input(){}
    input(std::string line)
    {
        std::string::size_type i;
        line.pop_back(); // remove newline
        i = line.find(' ');
        size = strtol(line.substr(0, i).c_str(), nullptr, 10);
        name = line.substr(i + 1);
    }
};

static
struct crush_map *build_map(int *ruleid,
                            int hostcount,
                            int blockcount);

static
void distribute(std::vector<std::vector<int>>& pg2host,
                int hostcount,
                int m,
                int pgcount);

static
void place(const char *prefix,
           const std::vector<std::vector<int>>& pg2host,
            const std::vector<input>& files,
            int k,
            int m,
            int pgcount);

static
void placeboth(const std::vector<std::vector<int>>& from,
               const std::vector<std::vector<int>>& to,
               FILE *in,
               int k,
               int m,
               int pgcount);

// cat files.txt | crush FROM TO K M
int main(int argc, const char *argv[])
{
    int frommapid, tomapid;
    struct crush_map *frommap, *tomap;
    long from, to, k, m, pg;

    if (argc < 6)
    {
        fprintf(stderr, "Usage: crush FROM TO K M PG <files.txt\n\n");
        fprintf(stderr, " FROM   # of hosts in the cluster frommap\n");
        fprintf(stderr, " TO     # of hosts in cluster tomap\n");
        fprintf(stderr, " K      # of blocks the file is broken into\n");
        fprintf(stderr, " M      # of blocks to distribute with straw2\n");
        fprintf(stderr, " PG     # of pgs to distribute\n");
        return 1;
    }

    from = strtol(argv[1], nullptr, 10);
    to = strtol(argv[2], nullptr, 10);
    k = strtol(argv[3], nullptr, 10);
    m = strtol(argv[4], nullptr, 10);
    pg = strtol(argv[5], nullptr, 10);

    std::vector<std::vector<int>> fromdist, todist;

    fprintf(stderr, "from: distributing pgs\n");
    distribute(fromdist, from, m, pg);
    fprintf(stderr, "to: distributing pgs\n");
    distribute(todist, to, m, pg);

#if 1
    fprintf(stderr, "placing files\n");
    placeboth(fromdist, todist, stdin, k, m, pg);
#else
    char *line;
    size_t size;
    ssize_t r;
    std::vector<input> files;
    line = nullptr;
    size = 0;
    while ((r = getline(&line, &size, stdin)) >= 0)
    {
        files.emplace_back(line);
    }
    fprintf(stderr, "from: placing files\n");
    place("from", fromdist, files, k, m, pg);
    fprintf(stderr, "to: placing files\n");
    place("to", todist, files, k, m, pg);
#endif
    return 0;
}

void distribute(std::vector<std::vector<int>>& pg2host,
                int hostcount,
                int m,
                int pgcount)
{
    int ruleno;
    struct crush_map *map;

    map = build_map(&ruleno, hostcount, m);

    // setup the workspace, make very large for no good reason
    char cwin[crush_work_size(map, 256)];
    crush_init_workspace(map, cwin);

    // * - weight[leaf] == 0x00000 == 0.0 always ignore
    // * - weight[leaf] == 0x10000 == 1.0 never ignore
    // * - weight[leaf] == 0x08000 == 0.5 ignore 50% of the time
    // * - weight[leaf] == 0x04000 == 0.25 ignore 75% of the time
    std::vector<__u32> weights(5000, 0x10000);

    for (int pg = 0; pg < pgcount; ++pg)
    {
        std::vector<int> result(m, -1);
        crush_do_rule(map, ruleno, pg, result.data(), m,
                weights.data(), weights.size(), cwin, nullptr);
        pg2host.push_back(std::move(result));
    }

    crush_destroy(map);
}

void place(const char *prefix,
           const std::vector<std::vector<int>>& pg2host,
           const std::vector<input>& files,
           int k,
           int m,
           int pgcount)
{
    for (auto& file : files)
    {
        unsigned fid, bid;
        MurmurHash3_x86_32(file.name.c_str(), file.name.length(), 111, &fid);
        bid = 0;
        for (int host : pg2host[fid % pgcount])
        {
            printf("%s,%u,node-0%02d,%s,%zu,%d,-\n",
                    file.name.c_str(), bid++, host + 6, prefix, file.size / k, m);
        }
    }
}


void placeboth(const std::vector<std::vector<int>>& from,
               const std::vector<std::vector<int>>& to,
               FILE *in,
               int k,
               int m,
               int pgcount)
{
    char *line;
    size_t size;
    ssize_t r;
    line = nullptr;
    size = 0;
    printf("#filename,bid,from,to,blocksize,m\n");
    while ((r = getline(&line, &size, in)) >= 0)
    {
        input file(line);
        unsigned fid, bid;
        MurmurHash3_x86_32(file.name.c_str(), file.name.length(), 111, &fid);

        auto& fromchain = from[fid % pgcount];
        auto& tochain = to[fid % pgcount];

        for (int bid = 0; bid < fromchain.size(); ++bid)
        {
          printf("%s,%u,node-0%02d,node-0%02d,%zu,%d\n",
                 file.name.c_str(),
                 bid,
                 fromchain[bid] + 6,
                 tochain[bid] + 6,
                 file.size / k,
                 m);
        }
    }
}


struct crush_map *build_map(int *ruleid, int hostcount, int blockcount)
{
      struct crush_map *map;
      struct crush_bucket *bucket, *root;
      struct crush_rule *rule;
      int i, rootno, bno;

      map = crush_create();

      int devid;
      int w = 0x10000;
      root = crush_make_bucket(map, CRUSH_BUCKET_STRAW2, CRUSH_HASH_RJENKINS1, TYPE_ROOT, 0, NULL, NULL);
      crush_add_bucket(map, 0, root, &rootno);
      for (devid = 0; devid < hostcount; ++devid)
      {
          bucket = crush_make_bucket(map, CRUSH_BUCKET_STRAW2, CRUSH_HASH_RJENKINS1, TYPE_HOST, 1, &devid,
               &w);
          crush_add_bucket(map, 0, bucket, &bno);
          crush_bucket_add_item(map, root, bno, w);
      }

      crush_finalize(map);

      ///////////////////////////////////////////////////////////////////////////////

      rule = crush_make_rule(3, 0, 0, blockcount, blockcount);
      crush_rule_set_step(rule, 0, CRUSH_RULE_TAKE, rootno, 0);
      crush_rule_set_step(rule, 1, CRUSH_RULE_CHOOSELEAF_INDEP, 0, TYPE_HOST);
      crush_rule_set_step(rule, 2, CRUSH_RULE_EMIT, 0, 0);

      *ruleid = crush_add_rule(map, rule, -1);
      return map;
}
```

C.4 analyze.pl

This analysis script reads the output of *placement.pl* C.2 or *crush.cpp* C.3 and generates basic stats about how data will be moved. Datapoints provided include the total number of bytes stored in the cluster (including erasure coding overhead), the number of blocks and bytes moved, the amount of data moved to existing nodes, and basic statistics and the standard deviation of bytes stored per node before and after the migration.

```
#!/usr/bin/env perl
# Copyright 2019 United States Government as represented by the Administrator
# of the National Aeronautics and Space Administration. All Rights Reserved.
use strict;
use warnings;

use Data::Dumper;
use File::Basename qw(basename);
use List::Util qw(min max sum);

# Usage: head files | ./placement.pl OPTIONS | ./analyze.pl
# #filename,bid,from,to,blocksize,m
# MOBRGB.A2000269.1605.005.2006269090714.jpg,0,node-009,node-009,40531,20
# MOBRGB.A2000269.1605.005.2006269090714.jpg,1,node-020,node-020,40531,20
# MOBRGB.A2000269.1605.005.2006269090714.jpg,2,node-024,node-028,40531,20
# MOBRGB.A2000269.1605.005.2006269090714.jpg,3,node-016,node-016,40531,20
# MOBRGB.A2000269.1605.005.2006269090714.jpg,4,node-006,node-034,40531,20

# ...
```

```perl
$Data::Dumper::Sortkeys = 1;
my $FONT_BOLD = `tput bold`;
my $FONT_RED  = `tput setaf 196`;
my $FONT_NONE = `tput sgr0`;

my %FLAGS = ();
for my $arg (@ARGV) {
    if ($arg =~ /--(\w+)(=([\w\-_=\s]+))?/) {
        if (defined $3) {
            $FLAGS{$1} = $3;
        } else {
            $FLAGS{$1} = 1;
        }
    }
}

my $DEBUG = exists $FLAGS{debug};

my %hosts = ();

my %fromblockcount = ();
my %frombytecount = ();
my %toblockcount = ();
my %tobytecount = ();

# num blocks moved to a host
my %moved = ();

my $total_block_count = 0;
my $total_bytes = 0;
my $numfiles = 0;
my $blocks_moved = 0;
my $bytes_moved = 0;
my $toold = 0;

while (<STDIN>) {
    # skip any line that starts with a #
    next if $_ =~ /^\s*#\s*.+$/;
    chomp;
    my ($path, $bid, $fromhost, $tohost, $bsize, $m) = split /,/;

    $hosts{$fromhost} = 1;
    $hosts{$tohost} = 1;

    $total_block_count += 1;
    $total_bytes += $bsize;
    $numfiles += 1 if $bid == 0;

    $fromblockcount{$fromhost} = &get(\%fromblockcount, $fromhost, 0) + 1;
    $frombytecount{$fromhost} = &get(\%frombytecount, $fromhost, 0) + $bsize;
    $toblockcount{$tohost} = &get(\%toblockcount, $tohost, 0) + 1;
    $tobytecount{$tohost} = &get(\%tobytecount, $tohost, 0) + $bsize;

    if ($fromhost ne $tohost) {
        $blocks_moved += 1;
        $bytes_moved += $bsize;
        $moved{$tohost} = &get(\%moved, $tohost, 0) + 1;
    }
}

# get all the block counts that went to a host that's in keys(%fromblockcount)
my %old = ();
for my $host (keys %hosts) {
    $old{$host} = &get(\%moved, $host, 0) if exists $fromblockcount{$host};
}
$toold = eval join('+', values(%old));

my $GiB_moved = sprintf("%.02f", $bytes_moved / (1024 ** 3));
my $GiB_moved_old = sprintf("%.02f", $toold / (1024 ** 3));

# Uncomment to provide a csv line with the stats instead of the human readable output
# my @usages = values %tobytecount;
# my $stddev = &stdev(\@usages);
# my $GiB_stddev = sprintf("%02f", $stddev / (1024 ** 3));
# print "$numfiles,$total_block_count,$total_bytes,$blocks_moved,$bytes_moved,$GiB_moved,$toold,
     $GiB_moved_old,$stddev,$GiB_stddev\n";
# exit 0;

###############################################################################
# REPORT
##
print "Cluster File/Block Count:     $numfiles / $total_block_count\n";
print "Cluster Total Bytes Used:     $total_bytes\n";

# stats about the rebalance
my $movedpct = sprintf("%.02f", ($blocks_moved / $total_block_count) * 100.0);
my $bytespct = sprintf("%.02f", ($bytes_moved / $total_bytes) * 100.0);

print "Rebalance Blocks Moved:       ${FONT_BOLD}${FONT_RED}${movedpct}%${FONT_NONE} ($blocks_moved)\n";
print "Rebalance Bytes Moved:        $bytespct% ($bytes_moved)\n";
print "Rebalance # Blocks to Old:    $toold\n";

# Distribution
print "Distribution Stats:\n";
&stats("Blocks Before", values %fromblockcount);
&stats(" Blocks After", values %toblockcount);
&stats(" Bytes Before", values %frombytecount);
&stats("  Bytes After", values %tobytecount);

###############################################################################

exit 0;

###############################################################################

# gets the value stored at key or default if key dne
sub get {
    my ($hashref, $key, $default) = @_;
    if (exists $hashref->{$key}) {
        return $hashref->{$key};
    }
    return $default;
```

```
}

sub stats {
    my $title = shift;
    my $min = min @_;
    my $max = max @_;
    my $mean = &mean(\@_);
    my $stddev = &stdev(\@_);
    my $stdev = sprintf "%0.02f", $stddev;
    my $psd = sprintf "%0.02f", ($stddev / sum @_) * 100;

    print "$title: $min\t$max\t$mean\t$stdev\t$psd%\n";
}

# https://edwards.sdsu.edu/research/calculating-the-average-and-standard-deviation/
sub mean{
        my($data) = @_;
        if (not @$data) {
                die("Empty arrayn");
        }
        my $total = 0;
        foreach (@$data) {
                $total += $_;
        }
        my $average = $total / @$data;
        return $average;
}
sub stdev{
        my($data) = @_;
        if(@$data == 1){
                return 0;
        }
        my $average = &mean($data);
        my $sqtotal = 0;
        foreach(@$data) {
                $sqtotal += ($average-$_) ** 2;
        }
        my $std = ($sqtotal / (@$data-1)) ** 0.5;
        return $std;
}
```

C.5 common.pl

We implemented each of the distribution methods using a common interface and placed them in *common.pl*. Note that to test CRUSH and CRUSH' we utilized *crush.cpp* C.3. This script holds all the method implementations we've tried to date, the good and the bad. We are able to quickly add new methods and ideas then use *placement.pl* C.2 to try against however many files or buckets we choose.

```
# Copyright 2019 United States Government as represented by the Administrator
# of the National Aeronautics and Space Administration.  All Rights Reserved.
use strict;
use warnings;

use Digest::MD5 qw(md5);
use Digest::MurmurHash3 qw(murmur128_x64);
use List::Util qw(max shuffle);
use File::Basename;

# defined in the test file
our $ERASURE_THRESHOLD;

# to test CRUSH or CRUSH' see ceph.cpp and update libcrush
# to either be unedited or with the patch applied
our %METHODS = (
    "hashchain", \&strategy_host,
    "hashchain-prime", \&strategy_npass_prime,
    "shuf-prime", \&strategy_shuf2,
    "ring", \&strategy_ring,
    "consistent", \&consistent
    "consistent-prime", \&consistent_prime,
);

# methods where the returned list are hostnames not hashes
our %HOST_METHODS = (
    "shuf", 1,
    "ring", 1,
    "consistent", 1,
);

our %HASHES = (
    "murmur3", \&hash_murmur3,
    "md5", \&hash_md5
);

# used by npass_prime_l
my $L = 8;

# must be set to one of the %HASHES values.
our $HASH_FUNCTION;

# must be defined by the test source file to be a hash of hostname to an array
# ref of unique host id and weight
our %HOSTS;

our @CRUSH_LN_16;

our $K;
our $M;

# flatten a cluster into a csv format of path, bid, to|from, m, host, link, bsize
sub flatten {
    my ($prefix, $cluster) = @_;
    for my $h (keys %$cluster) {
        for my $block (@{$cluster->{$h}}) {
            my ($path, $m, $bid, $bsize, $link) = @{$block};
            if (not $link =~ /^[[:print:]]+$/) {
                $link = unpack("h*", $link);
            }
            print "$path,$bid,$h,$prefix,$bsize,$m,$link\n";
        }
    }
}

# given the server's k, m, and erasure threshold return the number of blocks to
# be stored and the size of each block.
sub how_to_encode {
    my $size = shift;
    if ($size > $ERASURE_THRESHOLD) {
      my $topad = ($K - ($size % $K)) % $K;
      return ($M, int($size / $K) + $topad);
    } else {
        return (($M - $K) + 1, $size);
    }
}

sub hash_murmur3 {
    # murmur128_x64 returns 2 64 bit integers, convert to a byte string
    return pack "QQ", murmur128_x64(shift);
}

sub hash_murmur3_invert {
    # murmur128_x64 returns 2 64 bit integers, convert to a byte string
    return hash_invert(pack "QQ", murmur128_x64(shift));
}

sub hash_md5 {
    return hash_invert(md5(shift));
}

# take a 16 byte array convert to 2 64 bit ints add 1 to lower, and if overflow,
# add one to upper.
sub hash_add_one {
    my $link = shift;
    # convert back into 2 64 bit ints
    my ($b, $l) = unpack("QQ", $link);
    $l += 1;
    # overflow :-p
    if ($l == 0) {
        $b += 1;
    }
    return pack("QQ", $b, $l);
}

sub hash_invert {
  my $link = shift;
  # convert back into 2 64 bit ints
  my ($b, $l) = unpack("QQ", $link);
  return pack("QQ", $b ^ -1, $l ^ -1);
}
```

```
# no limitations, multiple blocks can live on the same host
sub strategy_none {
    my ($dist, $n, $path) = @_;
    my @chain = ($HASH_FUNCTION->(basename $path));
    for (my $i = 1; $i < $n; $i += 1) {
        push @chain, $HASH_FUNCTION->($chain[$i - 1]);
    }
    return \@chain;
}

# the existing strategy used in dishas
sub strategy_host {
    my ($dist, $n, $path) = @_;
    my %seen = ();
    my @chain = ($HASH_FUNCTION->(basename $path));
    $seen{&locate($dist, $chain[0])} = 1;
    for (my $i = 1; $i < $n; $i += 1) {
        my $next = $chain[$i - 1];
        my $node = "";
        do {
          $next = $HASH_FUNCTION->($next);
          $node = &locate($dist, $next);
        } while (exists $seen{$node});
        $seen{$node} = 1;
        push @chain, $next;
    }
    return \@chain;
}

# proposed two-pass alg
# this was the first attempt, but routinely has moved ~.5% more buckets then
# the npass approaches
sub strategy_2pass {
    my ($dist, $n, $path) = @_;
    my %seen = ();
    my @chain = ($HASH_FUNCTION->(basename $path));
    my @revisit = ();
    $seen{&locate($dist, $chain[0])} = 1;
    # first pass is to generate the hashes and assign all unique nodes possible
    for (my $i = 1; $i < $n; $i += 1) {
        my $next = $HASH_FUNCTION->($chain[$i - 1]);
        push @chain, $next;
        my $node = &locate($dist, $next);
        if (exists $seen{$node}) {
            push @revisit, $i;
        } else {
            $seen{$node} = 1;
        }
    }
    # second pass is only over the indicies that colided with another node
    for my $i (@revisit) {
        my $link = $chain[$i];
        # my $next = hash_invert($link);
        my $next = hash_add_one($link);
        my $node = "";
        do {
            $next = $HASH_FUNCTION->($next);
            $node = &locate($dist, $next);
        } while (exists $seen{$node});
        $seen{$node} = 1;
        $chain[$i] = $next;
    }
    return \@chain;
}

# multi-pass approach
# 1st pass same as above
# 2nd+ pass only hashes the revisit bids once then assigns if able
sub strategy_npass {
    my ($dist, $n, $path) = @_;
    my %seen = ();
    my @chain = ($HASH_FUNCTION->(basename $path));
    my @revisit = ();
    $seen{&locate($dist, $chain[0])} = 1;
    # first pass is to generate the hashes and assign all unique nodes possible
    for (my $i = 1; $i < $n; $i += 1) {
        my $next = $HASH_FUNCTION->($chain[$i - 1]);
        push @chain, $next;
        my $node = &locate($dist, $next);
        if (exists $seen{$node}) {
            push @revisit, $i;
        } else {
            $seen{$node} = 1;
        }
    }
    while (@revisit) {
        for (my $j = 0; $j <= $#revisit; $j += 1) {
            my $i = $revisit[$j];
            $chain[$i] = $HASH_FUNCTION->(hash_add_one($chain[$i]));
            my $node = &locate($dist, $chain[$i]);
            if (!exists $seen{$node}) {
                splice @revisit, $j, 1;
                $seen{$node} = 1;
            }
        }
    }
    return \@chain;
}

# multi-pass approach
# 1st pass same as above
# 2nd+ pass only hashes the revisit bids once then assigns if able
#
# This was an update to the first npass approach, I accidentaly made it add one
# every single round to the hash, this fixes that by only adding 1 to the first
# revisit hash
sub strategy_npass_prime {
    my ($dist, $n, $path) = @_;
    my %seen = ();
    my @chain = ();
    &impl_strategy_npass_prime($dist, $n, $path, \@chain, \%seen);
    return \@chain;
```

```
}

# seeing if altering the hash more than just adding one is worth the effort
# around a ~.1% difference and I think certain situations it can flip to single
# prime moving less blocks
sub strategy_npass_prime_invert {
    my ($dist, $n, $path) = @_;
    my %seen = ();
    my @chain = ($HASH_FUNCTION->(basename $path));
    my @revisit = ();
    $seen{&locate($dist, $chain[0])} = 1;
    # first pass is to generate the hashes and assign all unique nodes possible
    for (my $i = 1; $i < $n; $i += 1) {
        my $next = $HASH_FUNCTION->($chain[$i - 1]);
        push @chain, $next;
        my $node = &locate($dist, $next);
        if (exists $seen{$node}) {
            push @revisit, $i;
        } else {
            $seen{$node} = 1;
        }
    }

    # mutate the hashes we need to revist
    for my $i (@revisit) {
        $chain[$i] = hash_invert($chain[$i])
    }

    while (@revisit) {
        for (my $j = 0; $j <= $#revisit; $j += 1) {
            my $i = $revisit[$j];
            $chain[$i] = $HASH_FUNCTION->($chain[$i]);
            my $node = &locate($dist, $chain[$i]);
            if (!exists $seen{$node}) {
                splice @revisit, $j, 1;
                $seen{$node} = 1;
            }
        }
    }
    return \@chain;
}

# attempt to make npass_prime able to predict where the first `l` blocks are to
# ensure we can get a consensus allowing different erasure strategies
sub strategy_npass_prime_l {
    my ($dist, $n, $path) = @_;
    my @chain = ();
    my %seen = ();
    # first place the first l blocks with npass_prime
    &impl_strategy_npass_prime($dist, $L, $path, \@chain, \%seen);
    # place all other blocks
    &impl_strategy_npass_prime($dist, $n, $path, \@chain, \%seen);
    return \@chain;
}

# same as npass however it just maintains one chain of hashes instead of forking
# each colision by adding one
# consistently worse than npass, npass' and npass''
# This is consistently behaving worse then deriving new chains from the revisit
# hashes, not as bad as host but still ~2% more blocks moved then n-pass every
# time.
sub strategy_npass2 {
    my ($dist, $n, $path) = @_;
    my %seen = ();
    my @chain = ($HASH_FUNCTION->(basename $path));
    my @revisit = ();
    $seen{&locate($dist, $chain[0])} = 1;
    my $next = $chain[0];
    # first pass is to generate the hashes and assign all unique nodes possible
    for (my $i = 1; $i < $n; $i += 1) {
        $next = $HASH_FUNCTION->($next);
        push @chain, $next;
        my $node = &locate($dist, $next);
        if (exists $seen{$node}) {
            push @revisit, $i;
        } else {
            $seen{$node} = 1;
        }
    }
    while (@revisit) {
        for (my $j = 0; $j <= $#revisit; $j += 1) {
            my $i = $revisit[$j];
            $chain[$i] = $next = $HASH_FUNCTION->($next);
            my $node = &locate($dist, $next);
            if (!exists $seen{$node}) {
                splice @revisit, $j, 1;
                $seen{$node} = 1;
            }
        }
    }
    return \@chain;
}



# npass but not using a chain
sub strategy_npass3 {
    my ($dist, $n, $path) = @_;
    my %revisit = map { $_ => 1 } (0 .. ($n - 1));
    my %seen = ();
    my @chain = (1 .. $n);

    my $pass = 0;
    my $filename = basename $path;
    while (scalar(keys %revisit) > 0) {
        for my $bid (sort keys(%revisit)) {
            my $link = $HASH_FUNCTION->("${filename}\@${bid}.${pass}");
            my $node = &locate($dist, $link);
            unless (exists $seen{$node}) {
                $chain[$bid] = $link;
                $seen{$node} = 1;
                delete $revisit{$bid};
                }
            }
            $pass += 1;
        }
        return \@chain;
}

# use the filename hash to locate the first node, find that nodes index, then
# put all subsequent nodes on ($j + bid) % # hosts
sub strategy_ring {
    my ($dist, $n, $path) = @_;
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = sort keys %uniq;
    my @chain = ();
    my $node = &locate($dist, $HASH_FUNCTION->(basename $path));
    my $j = 0;
    for ($j = 0; $j <= $#hosts; $j += 1) {
        last if $hosts[$j] eq $node;
    }
    push @chain, $node;

    for (my $i = 1; $i < $n; $i += 1) {
        $j += 1;
        push @chain, $hosts[$j % scalar(@hosts)];
    }
    return \@chain;
}

# Hosts are listed by order they were added to the map and bucket 00000000 is
# assigned to the first host, then each subsequent host has the next bucket
# NOTE this leads to the placement group problem. As the mod value changes so
# does the initial node of sooooooo many objects
sub strategy_ring_mod {
    my ($dist, $n, $path) = @_;

    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = sort keys %uniq;

    # my @nodes = ();
    my @chain = ();

    # find the bucket
    # this is the definition of why placement groups exist as scalar(@hosts) increases the initial bucket
        location also changes
    my $j = unpack("Q", $HASH_FUNCTION->(basename $path)) % scalar(@hosts);
    # for my $b (keys(%$dist)) {
    #     if ($dist->{$b} eq $node) {
    #         my ($begin, $end) = unpack("LL", $b);
    #         push @chain, pack("L", $begin);
    #         last;
    #     }
    # }
    # push @nodes, $node;
    push @chain, $hosts[$j];
    for (my $i = 1; $i < $n; $i += 1) {
        $j = ($j + 1) % scalar(@hosts);
        # push @nodes, $hosts[$j];
        push @chain, $hosts[$j];
    }

    # # next block goes on $j + 1 % scalar(@hosts)
    # for (my $i = 1; $i < $n; $i += 1) {
    #     $j = ($j + 1) % scalar(@hosts);
    #     $node = $hosts[$j];
    #     push @nodes, $node;
    #     for my $b (keys(%$dist)) {
    #         if ($dist->{$b} eq $node) {
    #             my ($begin, $end) = unpack("LL", $b);
    #             push @chain, pack("L", $begin);
    #             last;
    #         }
    #     }
    # }
    # print($path . " -- " . Dumper(\@nodes) . "\n\n");
    return \@chain;
}


# emulates the CRUSH straw calculation for determining which nodes get a file
sub strategy_straw {
    my ($dist, $n, $path) = @_;

    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = keys %uniq;

    my @chain = ();
    my %used = ();

    for (my $i = 0; $i < $n; $i += 1) {
        my %straws = ();
        for my $host (@hosts) {
            my ($hid, $w) = @{$HOSTS{$host}};
            $straws{&draw_straw($path, $hid, $w, $i)} = $host unless exists $used{$host};
        }
        my $longest = max keys %straws;
        my $host = $straws{$longest};
        $used{$host} = 1;
        push @chain, $host;
    }
    return \@chain;
}

sub strategy_straw2 {
    my ($dist, $n, $path) = @_;

    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = keys %uniq;

    my @chain = ();
    my %used = ();
    for (my $i = 0; $i < $n; $i += 1) {
        my $max = "-inf";
        my $item = "";
        for my $host (@hosts) {
            unless (exists $used{$host}) {
                my ($hid, $w) = @{$HOSTS{$host}};
                my $straw = &draw_straw2($path, $hid, $w, $i);
                if ($straw > $max) {
                    $max = $straw;
                    $item = $host;
                }
            }
        }
        $used{$item} = 1;
        push @chain, $item;
    }
    return \@chain;
}

# like none but using straw2 instead of hash chains
sub strategy_straw2_none {
  my ($dist, $n, $path) = @_;

  my %uniq = map {$_ => 1} values(%$dist);
  my @hosts = keys %uniq;

  my @chain = ();

  for (my $i = 0; $i < $n; $i += 1) {
      my $max = "-inf";
      my $item = "";
      for my $host (@hosts) {
          my ($hid, $w) = @{$HOSTS{$host}};
          # print "$host $hid $w $i\n";
          my $straw = &draw_straw2($path, $hid, $w, $i);
          if ($straw > $max) {
              $max = $straw;
              $item = $host;
          }
      }
      push @chain, $item;
  }
  # print Dumper(\@chain);
  return \@chain;
}

# just draw straws once and sort the lengths descending
# this does not work, akin to the host strategy where a node being inserted
# shifts all future blocks.
sub strategy_straw2_once {
    my ($dist, $n, $path) = @_;

    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = keys %uniq;
    my @straws = ();

    for my $host (@hosts) {
        my ($hid, $w) = @{$HOSTS{$host}};
        my $straw = &draw_straw2($path, $hid, $w, 0);
        push @straws, [ $host, $straw ];
    }

    my @sorted = sort { $b->[1] <=> $a->[1] } @straws;

    my @chain = map { $_->[0] } @sorted;
    @chain = splice @chain, 0, $n;
    print STDERR Dumper(\@chain);
    return \@chain;
}

# attempt to reduce the number of straw drawings calculated.. this performs
# worse than host... and there isn't a way to do an npass on this so *shrug
sub strategy_straw_chain {
    my ($dist, $n, $path) = @_;

    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = sort keys %uniq;

    my @chain = ();
    my %straws = ();
    for my $host (@hosts) {
        my ($hid, $w) = @{$HOSTS{$host}};
        $straws{&draw_straw($path, $hid, $w, 0)} = $host;
    }
    #
    while (scalar(@chain) < $n) {
        my $longest = max keys %straws;
        push @chain, $straws{$longest};
        delete $straws{$longest};
    }

    return \@chain;
}

# instead of just not considering nodes that already have a block, emulate
# host_npass by keeping track of nodes we need to retry on a second pass
# had to alter the straw2 used in ceph to get this to work nicely
sub strategy_straw2_npass {
    my ($dist, $n, $path) = @_;

    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = sort keys %uniq;

    my @chain = ();
    # hosts that have been used so far
    my %used = ();
    # block id's that will be "" after a pass completes
    my %revisit = ();

    for (my $i = 0; $i < $n; $i += 1) {
        my $longest = "-inf";
        my $item = "";
        for my $host (@hosts) {
            my ($hid, $w) = @{$HOSTS{$host}};
            my $x = &draw_straw2($path, $hid, $w, $i);
            if ($x > $longest) {
              $longest = $x;
              $item = $host;
            }
        }

        if (exists $used{$item}) {
            $revisit{$i} = 1;
            push @chain, "";
        } else {
            $used{$item} = 1;
            push @chain, $item;
        }
    }

    my $pass = 1;
    while (%revisit) {
        my @toremove = ();
        for my $i (keys %revisit) {
            my $longest = "-inf";
            my $item = "";
            for my $host (@hosts) {
                my ($hid, $w) = @{$HOSTS{$host}};
                # my $x = &draw_straw2($path, $hid + $pass, $w, $i);
                if (!exists $used{$host}) {
                    $w *= ($pass ** 2.0);
                }
                my $x = &draw_straw2($path, $hid, $w, $i);
                if ($x > $longest) {
                    $longest = $x;
                    $item = $host;
                }
                # print "$host $x\n" if $path eq 'lads-data/MOD04_L2.A2020102.0545.061.2020102131724.hdf';
            }
            if (!exists $used{$item}) {
                delete $revisit{$i};
                $used{$item} = 1;
                $chain[$i] = $item;
            }
        }
        $pass += 1;
    }
    # print "took $pass passes\n";
    # print Dumper(\@chain);
    return \@chain;
}

# 2nd attempt: instead of changing the weight each pass. just don't consider all
# the nodes used in the previous pass.
sub strategy_straw2_npass2 {
    my ($dist, $n, $path) = @_;

    # at the end of each pass we remove all the hosts used on that pass. To
    # start it's all unique hosts.
    my %unused = map {$_ => 1} values(%$dist);
    # initiate an $n length array
    my @chain = (0..($n-1));
    # to begin with we need to still assign all indicies in @chain
    my @revisit = (0..($n-1));

    my $pass = 0;
    while ((my $len = scalar(@revisit)) > 0) {
        # foreach pass keep track of what hosts are now used and what indicies
        # in revisit are now satisfied.
        my %usedhosts = ();
        my @usedrevisits = ();
        for (my $j = 0; $j < $len; $j += 1) {
            my $i = $revisit[$j];
            my $longest = "-inf";
            my $item = "";
            for my $host (keys %unused) {
                my ($hid, $w) = @{$HOSTS{$host}};
                my $x = &draw_straw2($path, $hid, $w, $i);
                if ($x > $longest) {
                    $longest = $x;
                    $item = $host;
                }
            }

            if (!exists($usedhosts{$item})) {
                $usedhosts{$item} = 1;
                push @usedrevisits, $j;
                $chain[$i] = $item;
            }
        }

        # reverse the used indicies to make them descending order to make the
        # splice work.
        for my $j (reverse @usedrevisits) {
            splice @revisit, $j, 1;
        }

        # remove the newly used hosts from unused
        for my $host (keys %usedhosts) {
            delete $unused{$host};
        }
        $pass += 1;
    }

    # print "$pass passes needed for $path\n";
    # print Dumper(\@chain);
    # print Dumper(\%used);
    # print Dumper(\@revisit);
    return \@chain;
```

```
}

# 3rd idea - what if each pass we only assign items where only a single item
# wants the node. hmm much worse than straw2_npass2
sub strategy_straw2_npass3 {
    my ($dist, $n, $path) = @_;

    # at the end of each pass we remove all the hosts used on that pass. To
    # start it's all unique hosts.
    my %unused = map {$_ => 1} values(%$dist);
    # initiate an $n length array
    my @chain = (0..($n-1));
    # set of all block ids that have been assigned so far
    my %assigned = ();

    my $pass = 0;
    while (scalar(keys %assigned) < $n) {
        # each pass we map a host to an array of blocks that want it. Then only
        # assign nodes that want a single block.
        my %swipe = ();
        for (my $i = 0; $i < $n; $i += 1) {
            # skip index if already assigned
            next if exists $assigned{$i};
            my $longest = "-inf";
            my $item = "";
            for my $host (keys %unused) {
                my ($hid, $w) = @{$HOSTS{$host}};
                my $x = &draw_straw2($path, $hid, $w, $i + ($pass * $n));
                if ($x > $longest) {
                    $longest = $x;
                    $item = $host;
                }
            }

            if (!exists $swipe{$item}) {
                $swipe{$item} = [$i];
            } else {
                push $swipe{$item}, $i;
            }
        }

        # print Dumper(\%swipe);

        # if a host has a single block assigned to it, add to chain and remove
        # from sets for the next pass
        for my $host (keys %swipe) {
            my @blocks = @{$swipe{$host}};
            if (scalar(@blocks) == 1) {
                my $i = shift @blocks;
                $chain[$i] = $host;
                delete $unused{$host};
                $assigned{$i} = 1;
            }
        }
        $pass += 1;
    }
    # print "$pass passes needed for $path\n";
    # print Dumper(\@chain);
    # print Dumper(\%used);
    # print Dumper(\@revisit);
    return \@chain;
}

# https://www.spinics.net/lists/ceph-devel/msg21635.html
# STRAW
#  max_x = -1
#  max_item = -1
#  for each item:
#      x = random value from 0..65535
#      x *= scaling factor
#      if x > max_x:
#          max_x = x
#          max_item = item
#  return item
#
# STRAW2
#  max_x = -1
#  max_item = -1
#  for each item:
#      x = random value from 0..65535
#      x = ln(x / 65536) / weight
#      if x > max_x:
#          max_x = x
#          max_item = item
#  return item

# ignores weights right now which is fine since all are of = weight to us
sub draw_straw {
    my ($path, $hostid, $w, $bid) = @_;
    my $hash = $HASH_FUNCTION->(basename($path) . "\@$bid\@$hostid");
    return unpack("Q", $hash) * $w;
}

# https://github.com/ceph/ceph/pull/20196/files#diff-520
     ee7de08c6da22b920d4c8892b09a71e7422591f7d177f7b236e5e7997f201L345
# sub draw_straw2 {
#      my ($path, $hostid, $w, $bid) = @_;
#      my $hash = $HASH_FUNCTION->(basename($path) . "\@$bid\@$hostid");
#      my $x = ((unpack("Q", $hash)) % 65536);
#      if ($x == 0) {
#           # https://github.com/yaozongyou/ceph/blob/4faadeae1122387e3aaf0b4623e0246d09ed5ead/src/crush/
     crush_ln_table.h#L27
#           $x = 0x1000000000000;
#      } else {
#           $x = log($x / 65536);
#      }
#      # https://github.com/yaozongyou/ceph/blob/4faadeae1122387e3aaf0b4623e0246d09ed5ead/src/crush/mapper.
     c#L350
#      $x -= 0x1000000000000;
#      return $x / $w;
# }
# we stole the crush_ln function and table from libcrush to generate all 65536
# possible values
# require "crushln.pl";
# sub draw_straw2 {
#     my ($path, $hostid, $w, $bid) = @_;
#     my $hash = $HASH_FUNCTION->(basename($path) . "\@$bid\@$hostid");
#     my $x = unpack "S", $hash; # S is an unsigned 16 bit short
#     return $CRUSH_LN_16[$x] / $w;
# }

#
sub impl_strategy_npass_prime {
    my ($dist, $n, $path, $chain, $seen) = @_;
    my @revisit = ();
    if (scalar(@$chain) == 0) {
      push @$chain, $HASH_FUNCTION->(basename $path);
      $seen->{&locate($dist, $chain->[0])} = 1;
    }
    # first pass is to generate the hashes and assign all unique nodes possible
    for (my $i = scalar(@$chain); $i < $n; $i += 1) {
        my $next = $HASH_FUNCTION->($chain->[$i - 1]);
        push @$chain, $next;
        my $node = &locate($dist, $next);
        if (exists $seen->{$node}) {
            push @revisit, $i;
        } else {
            $seen->{$node} = 1;
        }
    }

    # mutate the hashes we need to revist
    for my $i (@revisit) {
        $chain->[$i] = hash_add_one($chain->[$i])
    }

    my $pass = 0;
    while (@revisit) {
        $pass += 1;
        for (my $j = 0; $j <= $#revisit; $j += 1) {
            my $i = $revisit[$j];
            $chain->[$i] = $HASH_FUNCTION->($chain->[$i]);
            my $node = &locate($dist, $chain->[$i]);
            if (!exists $seen->{$node}) {
                splice @revisit, $j, 1;
                $seen->{$node} = 1;
            }
        }
    }
}

# wow this is bad 96% data moved on 1000 input files.
sub strategy_shuf {
    my ($dist, $n, $path) = @_;
    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = sort keys %uniq;
    srand unpack("Q", $HASH_FUNCTION->(basename $path)) % 65536;
    @hosts = shuffle @hosts;
    my @chain = @hosts[0..($n - 1)];
    return \@chain;
}

# standard shuffle alg but instead of using random num generation use a hash
sub shuf {
    my ($arr, $seed) = @_;
    for (my $i = scalar(@$arr) - 1; $i > 0; $i -= 1) {
        my $j = int((unpack("S", $HASH_FUNCTION->("$seed.$i")) / 65535) * $i);
        # my $j = int((unpack("I", $HASH_FUNCTION->("$seed.$i")) / (1<<32)) * $i);
        # print "swapping $i with $hash / 65536 = $j\n";
        my $tmp = $arr->[$i];
        $arr->[$i] = $arr->[$j];
        $arr->[$j] = $tmp;
    }
    return $arr;
}

# don't use srand/rand use the hash as the random num
sub strategy_shuf2 {
    my ($dist, $n, $path) = @_;
    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = sort keys %uniq;
    my %unused = map {$_ => 1} @hosts;
    my %revisit = map { $_ => 1 } (0..($n-1));
    my $pass = 0;
    my @chain = (1..$n);
    while (%revisit) {
        for (my $bid = 0; $bid < $n; $bid += 1) {
            if (exists $revisit{$bid}) {
                my @cpy = @hosts;
                &shuf(\@cpy, basename($path) . "\@$bid.$pass");
                my $host = shift @cpy;
                if (exists $unused{$host}) {
                    $chain[$bid] = $host;
                    delete $unused{$host};
                    delete $revisit{$bid};
                }
            }
        }
        if (scalar keys %revisit == 1) {
            my ($k) = keys %revisit;
            ($chain[$k]) = keys %unused;
            delete $revisit{$k};
        }
        $pass += 1;
    }
    return \@chain;
}

# shuf2 without revisiting, worse than shuf2
sub strategy_shuf3 {
    my ($dist, $n, $path) = @_;
    # get a list of all hosts and assign unique hostid to each
    my %uniq = map {$_ => 1} values(%$dist);
    my @hosts = sort keys %uniq;
    my %unused = map {$_ => 1} @hosts;
    my @chain = (1..$n);
    for (my $bid = 0; $bid < $n; $bid += 1) {
        my @cpy = sort keys %unused;
        &shuf(\@cpy, basename($path) . "\@$bid");
        my $host = shift @cpy;
        $chain[$bid] = $host;
        delete $unused{$host};
    }
    return \@chain;
}

sub consistent_indexof {
    my ($arr, $key) = @_;
    # do a linear for right now this can be rewritten as a binary search later
    for my $i (0..$#$arr) {
        if ($arr->[$i]->[0] >= $key) {
            return $i;
        }
    }
    # if our key is greater than any host in the array then we wrap around to
    # position 0
    return 0;
}

# use the hash to generate the order, then set each's key as 2^64/numhosts*50
# this was a terrible idea, while it improves stddev you move way too much
sub build_hashring_even {
    my ($dist, $n, $bid) = @_;
    my $name = defined($bid)? ".consistent_hash_$bid" : ".consistent_hash";
    my $mapping = $dist->{$name};
    unless (defined($mapping)) {
        my %uniq = map {$_ => 1} values(%$dist);
        my @hosts = keys %uniq;
        my @builder = ();
        for my $host (@hosts) {
            for my $i (0..100) {
                my $hostname = defined($bid)? "$host\@${i}_$bid" : "$host\@$i";
                my $key = unpack("Q", $HASH_FUNCTION->($hostname));
                push @builder, [$key, $host];
            }
        }
        # our ring goes from 0 -> 2^64
        my @mapping = sort {$a->[0] <=> $b->[0]} @builder;
        my $step = int(2**64 / scalar(@mapping));
        for my $i (0..$#mapping) {
            $mapping[$i]->[0] = $i * $step;
        }
        $dist->{$name} = \@mapping;
        $mapping = \@mapping;
    }
    return $mapping;
```

```
}

# standard consistent hashing circle, add each node $n times, if $bid
# exists then create a special ring just for that block index
sub build_hashring {
    my ($dist, $n, $bid) = @_;
    my $name = defined($bid)? ".consistent_hash_$bid" : ".consistent_hash";
    my $mapping = $dist->{$name};
    unless (defined($mapping)) {
        my %uniq = map {$_ => 1} values(%$dist);
        my @hosts = keys %uniq;
        my @builder = ();
        for my $host (@hosts) {
            # even at 200 placements we still have a worse std dev compared to
            # CRUSH and Hashchain family algs.
            for my $i (0..$n) {
                my $hostname = defined($bid)? "$host\@${i}_$bid" : "$host\@$i";
                my $key = unpack("Q", $HASH_FUNCTION->($hostname));
                push @builder, [$key, $host];
            }
        }
        # our ring goes from 0 -> 2^64
        my @mapping = sort {$a->[0] <=> $b->[0]} @builder;
        $dist->{$name} = \@mapping;
        $mapping = \@mapping;
    }
    return $mapping;
}

sub consistent_prime {
    my ($dist, $n, $path) = @_;
    my $mapping = &build_hashring($dist, 100);

    # find the index of all parts
    my $name = basename $path;
    my @keys = ();
    for my $bid (0..$n-1) {
        push @keys, unpack("Q", $HASH_FUNCTION->("$name\@$bid"));
    }

    my @chain = ();

    # 1  each swipe remove all used hosts from the hashring
    # 2  each blockid generates a different hashring "$hostname@$vhostnum_$bid"
    # 0  don't mutate the ring and just skip entries that point to used hosts
    #
    # 1 is worse than 2 which is worse than 0 ... hmmm
    my $SWIPE = 0;
    if ($SWIPE == 1) {
        # hash of all available hosts
        my %avail = map {$_ => 1} values(%$dist);
        # block ids to revisit
        my %revisit = map { $_ => 1 } (0..($n-1));
        while (%revisit) {
            # create a mapping where only available hosts are used
            my @swipemapping = ();
            for my $pair (@$mapping) {
                if (exists($avail{$pair->[1]})) {
                    push @swipemapping, $pair;
                }
            }
            # iterate through bids in order
            for my $bid (0..$n-1) {
                if (exists($revisit{$bid})) {
                    my $pos = &consistent_indexof(\@swipemapping, $keys[$bid]);
                    my $host = $swipemapping[$pos]->[1];
                    if (exists($avail{$host})) {
                        delete $avail{$host};
                        delete $revisit{$bid};
                        push @chain, $host;
                    }
                }
            }
        }
    } elsif ($SWIPE == 2) {
        my %used = ();
        # We generate a new hashring for every bid
        for my $bid (0..$n-1) {
            $mapping = &build_hashring($dist, 100, $bid);
            my $key = $keys[$bid];
            my $offset = 0;
            my $i = &consistent_indexof($mapping, $key);
            while (exists($used{$mapping->[($i + $offset) % scalar(@$mapping)]->[1]})) {
                $offset += 1;
            }
            my $host = $mapping->[($i + $offset) % scalar(@$mapping)]->[1];
            push @chain, $host;
            $used{$host} = 1;
        }
    } else {
        # no swipe
        my %used = ();
        for my $key (@keys) {
            my $offset = 0;
            my $i = &consistent_indexof($mapping, $key, 0);
            while (exists($used{$mapping->[($i + $offset) % scalar(@$mapping)]->[1]})) {
                $offset += 1;
            }
            my $host = $mapping->[($i + $offset) % scalar(@$mapping)]->[1];
            push @chain, $host;
            $used{$host} = 1;
        }
    }
    return \@chain;
}

1;
```