Randomized Algorithms for Dynamic Storage Load-Balancing

Liang Liu[†] Lance Fortnow[†] Jin Li[‡] Yating Wang[†] Jun Xu[†]
Georgia Institute of Technology[†], Microsoft[‡]
lliu315@gatech.edu {fortnow,jx}@cc.gatech.edu jinl@microsoft.com carmen78xy@gmail.com

Abstract

In this work, we study a challenging research problem that arises in minimizing the cost of storing customer data online for reliable access in a cloud. It is how to near-perfectly balance the remaining capacities of all disks across the cloud system while adding new file blocks so that the inevitable event of capacity expansion can be postponed as much as possible. The challenges of solving this problem are twofold. First, new file blocks are added to the cloud concurrently by many dispatchers (computing servers) that have no communication or coordination among themselves. Though each dispatcher is updated with information on disk occupancies, the update is infrequent and not synchronized. Second, for fault-tolerance purposes, a combinatorial constraint has to be satisfied in distributing the blocks of each new file across the cloud system. We propose a randomized algorithm, in which each dispatcher independently samples a blocks-to-disks assignment according to a probability distribution on a set of assignments conforming to the aforementioned combinatorial requirement. We show that this algorithm allows a cloud system to near-perfectly balance the remaining disk capacities as rapidly as theoretically possible, when starting from any unbalanced state that is correctable mathematically.

Categories and Subject Descriptors

C.1.2 [Multiple Data Stream Architectures]
E.2 [Data Storage Representations]

Keywords

load-balancing; diversity requirement; load distribution

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.

ACM 978-1-4503-4525-5/16/10. . . $15.00.
http://dx.doi.org/10.1145/2987550.2987572

1. Introduction

Cloud service providers such as Amazon and Microsoft have to store massive amounts of customer data in their data centers. Customers expect to have reliable access to their data files at any time. However, the storage devices and computers that host the data, and the networks that transport the data, are prone to disruptions caused by power failures, scheduled maintenances, upgrades and hardware malfunctions. In addition, the storage devices and computers may need to be temporarily taken out of service for software patch, new features installation and configuration change. A portion of the devices and computers may become hot spots, leading to long service latency. Human operator errors may also render some computers and/or networks temporarily inoperable.

We maintain reliable access in the presence of these disruptions typically through a combination of two risk control measures. One is through erasure coding, that is, to divide a data stream (aggregation of customer files) into b blocks, and then encode them into $b+b'$ blocks (by adding b' blocks of redundancies) so that missing one or two such blocks due to disruptions will not prevent the file from being decoded and delivered to the customer [17]. The other is to strategically distribute these (coded) blocks across the data center (e.g., to place them on different racks of storage servers, or even different zones serviced by distinct power units) so that a typical service disruption event (e.g., failure of a power unit that affects a zone, failure of a top of rack switch that affects a rack) may affect the access to at most one or two such (coded) blocks [20]. With the combination of these two risk control measures, we can greatly reduce the risk of customers having their file access interrupted due to disruptions.

A cloud service provider typically would like to perfectly balance the load across the disks so that it has maximum flexibility in accommodating new customers and their data. Due to the constraints in distributing blocks of files across a data center, which we will elaborate shortly, it is impossible to perfectly balance the load across all disks. Therefore, how efficiently the disk space is managed is typically measured by how close the average disk occupancy ratio of the cloud storage can approach a pre-determined "safe" upper limit (say 80%), before the occupancy ratio of a disk exceeds this limit.

1.1 The Load-Balancing Problem

In this work, we first tackle a challenging research question that arises in our attempts at keeping disk load as balanced as possible under the various constraints a data center has to work with in distributing coded blocks of file across its computers/disks. One key constraint is the diversity requirement, which can be mathematically formulated as follows. As we will elaborate in Section 2, the disk space of a data center can be divided into mn partitions – called cells – organized as an $m \times n$ matrix. Upon the arrival of a new fixed-size file extent, each of its k coded blocks needs to be added to one of these cells. The diversity requirement is that no cell can take in more than one such block, and no two cells that take in a block can be on the same row or column of the matrix. As we will explain shortly, this diversity requirement ensures that the disruptive events, which typically affect one entire row or column at a time, will not prevent any file block from being decoded and accessed, with overwhelming probabilities.

With the diversity requirement, this problem seems to call for a (deterministic) greedy solution, in which we try to place k blocks in a diverse set of k cells that collectively have the smallest overall occupancy ratios. However, another requirement, known as "distributed independent dispatching", renders such a greedy solution unsuitable. In a cloud environment, new file extents are added by a large number of front-end computing servers that interact with customers, which we call dispatchers. "Distributed independent dispatching" means that no communication or coordination is required among dispatchers when they add new file extents to the disks. In addition, in the case of at least one major cloud provider, dispatchers obtain readings on the loads (occupancy ratios) of all disks (cells) in a data center quite infrequently, usually once a day. Such infrequent reporting rules out any coordination among dispatchers through disk occupancy readings.

The greedy solution does not fulfill the "distributed independent testing" requirement. With a deterministic greedy solution, the actions of dispatchers are to a large degree synchronized in the sense they all try to add blocks to the same set of cells that are, according to the once-a-day load report, relatively lightly loaded. Due to this synchronization, these (relatively) lightly loaded cells will take in so many new blocks from the dispatchers that they could quickly become grossly overloaded.

1.2 A Randomized Approach

A randomized algorithm solution, on the other hand, generally does not suffer from this undesirable synchronization. Therefore, our research effort has been focused on finding and refining such a randomized solution. A randomized algorithm for solving this load-balancing problem generally works as follows. Each dispatcher, upon receiving a disk occupancy report, precomputes a probability distribution on a set of possible blocks-to-cells assignments conforming to the diversity requirement. Later on, when a dispatcher has to add a new file extent to the data center, it randomly samples a blocks-to-cells assignment according to the computed distribution, and dispatches the blocks accordingly. Note it is important to do this in two steps, since the first step – computing the desired distribution – which is done offline, could take a long time (typically minutes). The second step – making random assignments based on the precomputed probability distribution – on the other hand, must be finished in milliseconds in order not to "freeze" the cloud service.

Our main research problem is therefore to identify and compute a suitable probability distribution, that, after a large number of random blocks-to-cells assignments are made – independently by many dispatchers without any coordination among themselves – according to the distribution, the residual spaces of all disks in the cloud will be made very close to one another, even when they are far apart to start with. Such a probability distribution, however, turns out to be quite hard to identify and to compute. A major contribution of this work is to identify and efficiently compute such a distribution. We will show that our randomized load-balancing scheme with such a distribution can balance the disk loads across different cells almost as perfectly as the ideal scheme, under the diversity constraint.

Readers might wonder whether the uniform distribution – sampling blocks-to-cells assignments uniform randomly from the set of all assignments that conform to the diversity requirement – would do the trick. It is however not suitable for this task because the uniform distribution carries no corrective power: If the disk occupancy ratios across cells are very uneven to start with, or all of sudden become uneven – due to a row or column of cells being taken down for repair or maintenance for an extended period of time – uniform distribution is not able to correct the situation. This unevenness will persist and may leave many cells severely underutilized while some cells become completely full.

1.3 Summary of Contributions

In this work, we have made four contributions. First, we propose a distributed randomized load-balancing scheme that can near-perfectly restore the load-balancing among the disks, when they are unbalanced to start with, under all aforementioned operational constraints of the cloud system. Second, we propose a new sweeping algorithm that, after the load becomes balanced, perfectly maintains this balance, despite the distributed uncoordinated nature in which the assignments are made. Third, we identify a probability distribution suitable for making the randomized load assignments. This probability distribution can be computed via a classical matrix decomposition process. The best algorithm for carrying out this decomposition however has a very high computational complexity. Our fourth contribution is to design a much more efficient algorithm for computing this probability distribution.

The rest of the paper is organized as follows. In Section 2, we elaborate on the dispatching constraints and formulate the problem by introducing a system model in accordance with those constraints. In Section 3, we propose our main dispatching scheme that consists of two major stages – a weighted randomized assignment stage and a sweeping stage. In Section 4, we offer a more efficient option for computing the probability distribution during the first stage of our scheme. In Section 5, we evaluate the performance of our scheme against the uniform randomized algorithm using extensive simulations based on ballpark parameters provided by a tier-1 cloud service provider. Section 6 discusses prior works related to our paper. The last section concludes the paper.

2. System Environment and Problem Statement

We explained earlier that the disk space of a data center is logically partitioned into mn cells that are organized as an $m \times n$ matrix, and the diversity requirement that k blocks that comprise of an extent need to be assigned to distinct rows and columns of the matrix. In this section, we describe the system environment of a data center, emphasizing aspects important to this work, including hardware and software failures, failure tolerance and recovery measures, and related costs and tradeoffs. Then we state the problem precisely and describe the constraints that must be satisfied by any viable solution to it.

2.1 System Environment

2.1.1 Hardware Failure Zones

A top tier cloud service provider has typically multiple service regions, each of which contains multiple data center facilities. Within a single data center facility, the storage servers, network equipments and power supply units are often further grouped hierarchically. At the top level of this hierarchy are so-called *hardware failure zones*. Each hardware failure zone consists of multiple co-located aisles of storage server racks served by a single power distribution unit (PDU), which employs transformers to convert down the voltage through feeder circuits. A single hardware failure will typically affect at most one hardware failure zone, and hence its name. Each hardware failure zone corresponds to a row of cells in the $m \times n$ cloud matrix, as shown in Figure 1.

Further down the hardware grouping hierarchy, a hardware failure zone is divided into aisles and racks, in which a rack of servers is usually served by a common rack-mount power distribution unit (rPDU) and top-of-rack (TOR) switch. The hierarchical organization and design of the cloud data centers recognizes the fact that in a complex operating environment, every hardware component can fail. The most common type of failure is that of a single disk or server, which could happen almost every hour. The failed disk or server may be left unrepaired if the cost of repair service outweighs the disk capacity recouped through the repair.

The failure of a component that affects a rack of servers or more (e.g., TOR or rPDU or even PDU) is less common, but cannot not be ignored, since the recouped disk capacity more than adequately justifies the cost of repair. However, even in such a case, it is desirable that the failure can be tolerated for a relatively long time interval, e.g., 7 days. The reason is that rapid on-site response (e.g., 4 hours) and 24x7 emergency service translate into a very expensive service contract and are labor intensive. The cost of the repair and the associated service contract will be far cheaper if the Mean Time to Repair (MTTR) can be stretched longer. However, longer MTTR results in higher level of disk load imbalance after the repair, which is more difficult and takes longer to correct. One goal of our load-balancing scheme is therefore to restore the balance as quickly as theoretically possible.

2.1.2 Software Update Zones

In addition to hardware failures, the servers and networks in a data center may also experience outages due to software issues. A common software issue is the need to upgrade server softwares, apply security patches, or change the servers/networks configuration. Since the servers and networks undergoing such updates may be temporarily out of service, a cloud storage system is typically divided to multiple equal-sized (in terms of disk storage volume or network bandwidth) logical groups called *software update zones*. These updates are then performed on one software update zone, at a time, so that continuous cloud service can be provided to customers by other zones not undergoing the process at the moment, as we will explain shortly. Another reason for such "zone rotation" is that incorrectly applied update or configuration change could render all affected servers inoperable, and zone rotation firewalls the damage within only the affected zone.

Each software update zone corresponds to a column of cells in the cloud matrix, as shown in Figure 1. In other words, the partitioning of the data center into software update zones (columns) are designed to be "perpendicular" to that into the hardware failure zones (rows). The reason for this perpendicularity is that it minimizes the impact of the simultaneous occurrence of a hardware failure and a software failure. Consider the totally opposite arrangement in which software updates are applied to an entire data center facility (consisting of multiple hardware failure zones as explained earlier). Suppose a software update takes an entire data center facility out of service. In the meantime, another hardware failure zone may experience a concurrent failure due to a hardware issue. Since these two simultaneous failures impact multiple hardware failure zones, they are beyond the redundancy level designed for the erasure correction coding.

2.1.3 Erasure Correction Coding

Cloud service providers aim to provide customers with uninterrupted access to their data files at all times, even though the underlying hardware and software fail all the time. This goal is usually achieved using erasure correction coding as follows. The data files written by customers are aggregated and appended to a blob store, in the form of extents. Whenever an extent reaches a certain size (say 3GB), it is sealed and erasure-coded. For example, for the proprietary LRC code employed by Microsoft Windows Azure Storage [17], each extent is divided into 14 data blocks, and 4 additional code blocks are computed. This $14+4$ code is designed to withstand arbitrary 3 failures (i.e., 3 blocks becoming lost or unaccessible) and 85% of 4 failures. The advantage of erasure correction coding is significant capacity saving comparing with typical replication method. To tolerant 3 failures, at least 4 replicas will be needed to kept in the data center while using replication to do fault-tolerance, and the total data amount will be four times of original data amount, which is much larger than $\frac{18}{14}$ by erasure coding [24].

The diversity requirement is simply that the data and code blocks of an extent be placed on distinct software upgrade zones (rows) and distinct hardware failure zones (columns) of the cloud matrix. With the diversity requirement and the LRC coding, at least 4 simultaneous hardware and software failures have to happen – an event with vanishingly small probability – for more than 3 original or coded blocks to be unaccessible that is the necessary condition for the extent to be unaccessible. In this operating environment, the performance of the cloud storage is only slightly degraded if only one data or code block becomes unaccessible. However, if two or more blocks become unaccessible, the dispatcher that is in charge of the extent will need to devote network bandwidth and storage capacity – and hence a slight degradation of the storage system performance – to actively repair and reconstruct the unaccessible blocks so that even if the failed server never comes back to service, the data of the customers are not eventually lost.

2.1.4 A Cell in the Cloud Matrix

As shown in Figure 1, a cell is simply the intersection of a hardware failure zone (row) and a software update zone (column). It typically consists of a full or a half rack of storage servers. We assume that, within a cell, the storage loads of servers can be perfectly balanced, allowing a block to be inserted even if only one server has a disk that is not completely full. This assumption is quite realistic for three reasons. First, a server can certainly add a block to any of its own disks that are not completely full. Second, communications between servers on a rack need only go through the top-of-rack (TOR) switch that has very high network bandwidth (per server) and very low delay, allowing servers to maintain up-to-date load conditions of one another – and hence to route a new block to the server that is least loaded, without degrading the system performance. Third, when load becomes unbalanced within a rack, the overhead is also relatively low of moving blocks around the servers to restore the local load-balance. Note that such movements within the same cell will not cause any violation of the diversity requirement. Therefore, throughout the rest of the paper, we worry only about load-balancing among the cells, not within a cell.

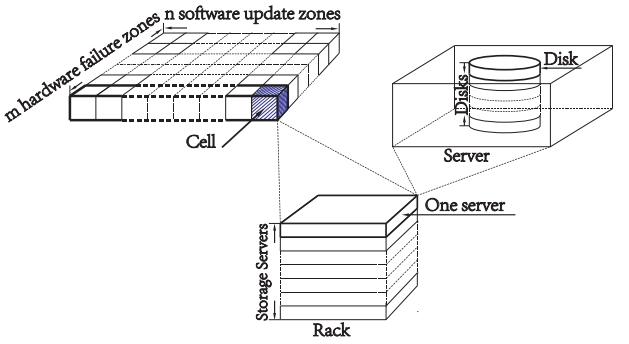

Figure 1. The Logical Structure of the Cloud System

For the convenience of presentation, in our load-balancing scheme design, we assume every cell has the same total storage capacity. In reality, cells often have different total capacities. In particular, the cloud provider usually purchases one "row" of servers (i.e., a hardware failure zone) at a time on a mounted truck together with all the racks, network wiring, switches, and power wiring, and "rows" purchased later in time typically have larger total disk capacities, but at similar costs; This bigger-bang-for-the-bucks effect indeed provides a strong motivation for achieving near-perfect load-balancing, which postpones the purchase of a new truckload of storage servers for as long as possible. In Section 3.2, we will prove this convenient homogeneous cell capacity assumption is "innocuous": It does not make our load-balancing problem any simpler or our solution any weaker.

3. Cloud Storage Load-Balancing Problem and Solution

In this section, we mathematically formulate the load-balancing problem in Section 3.1. In Section 3.2, we do the problem reduction which maps any cloud system to a corresponded cloud system with uniform cell capacity. Based on the reduction, we introduce our randomized solution in Section 3.3.

3.1 Problem Statement

Mathematically, the set of assignments that satisfies the diversity requirement corresponds precisely to the set of $m \times n$ k-matching matrices. In a k-matching matrix, all but k elements take value zero; These k nonzero elements all take value 1 and are distributed across k different rows and k different columns. The name "k-matching" is derived from the fact when such a matrix is viewed as the incidence matrix

of a bipartite graph, the corresponding bipartite graph is a matching of size k. Given an $m \times n$ k-matching matrix, the corresponding assignment is to dispatch the k blocks of data to the k corresponding cells in the $m \times n$ cloud matrix. Let $L = \{l_{ij}\}_{m\times n}$ be the load matrix of the cloud system in the sense each l_{ij} is the total storage load of the cell on the i_{th} row and j_{th} column of the cloud matrix. Assume that the size of each block is 1 (unit). Then after such an assignment, the load matrix of the cloud is simply incremented by the corresponding k-matching matrix.

Our load-balancing problem can be characterized mathematically as follows. Upon the arrival to a dispatcher of a new extent – consisting of k coded blocks – that needs to be added to the data center, the dispatcher needs to compute a k-matching matrix – independent of all other dispatchers – that is used to assign these blocks to the corresponding cells. Each such decision will be informed by only the starting value of the load matrix L, in which loads of different cells could be very different than one another – and infrequent subsequent updates on the values of L. Yet, after a certain number of such assignments are made – which ideally are close to the minimum needed – the loads of cells in the resulting load matrix are very close to one another.

3.2 Problem Reduction

In this section, we prove by reduction that the homogenous cell capacity assumption is "innocuous". Let $V = \{v_{ij}\}_{m\times n}$ be the capacity matrix of the cloud system in the sense each v_{ij} is the total capacity of the cell on the i_{th} row and j_{th} column of the cloud matrix. Note the *actual* goal of our load-balancing algorithm, with or without this assumption, is to maximize $\min_{i,j}(v_{ij} - l_{ij})$ (the minimum remaining capacity), which is equivalent to, under this assumption, the previously stated goal of minimizing $\max_{i,j} l_{ij}$. In addition, it will become clear shortly that the actions taken by our randomized load-balancing scheme are determined, except for the outcomes of coin tosses, by only the remaining capacity matrix $V - L$. Therefore, the above assumption is innocuous as long as we can show that given any capacity-load scenario (V, L) that violates this assumption (i.e., v_{ij} values may not be identical), we can reduce it to a scenario (V', L') that conforms to this assumption (i.e., $V' = v \cdot \mathbb{1}_{m\times n}$), such that $V - L = V' - L'$, as follows. Define $v \equiv \max_{i,j} v_{ij}$ and $L' \equiv L + v \cdot \mathbb{1}_{m\times n} - V$. We have $V' - L' = v \cdot \mathbb{1}_{m\times n} - (L + v \cdot \mathbb{1}_{m\times n} - V) = V - L$.

3.3 Our Randomized Solution

Our distributed randomized solution consists of two stages. During the first stage, each dispatcher, starting with any initial cloud load distribution that can be quite uneven, assigns blocks to cells independent of one another, according to a distribution that is intended to balance the load over time. Once the load becomes fairly balanced, the scheme enters the second stage, which we refer to as the "cruising stage". In the cruising stage, a variant of the uniform distribution specially crafted to minimize the (local) differences among the amount of loads a dispatcher imposes on each cell – instead of the above distribution – is used by dispatchers to maintain this load-balancing while adding new extents to the cloud system.

3.3.1 The Load-Balancing Stage

The pseudo-code of the distributed randomized algorithm executed by a dispatcher is very simple and is shown in Algorithm 1. Whenever the dispatcher needs to make a blocks-to-cells assignment, it samples a matrix Φ from a set of $m \times n$ k-matching matrices $\{\Phi_1, \Phi_2, ...\Phi_q\}$, according to a pre-computed distribution $\mathbb{P}[\Phi = \Phi_i] = p_i$, for $i = 1, 2, ..., q$, and makes the assignment accordingly. Since q is quite small, as will be explained shortly, the (online) dispatching step can be done in nanoseconds using a pre-computed table stored in SRAM, or in microseconds when DRAM is used instead. Note that the successive sampling operations performed by a dispatcher, and hence the successive assignments, are independent of one another, and that the actions of dispatchers are independent of one another. This statistical independence, among the assignments made both at a dispatcher and across different dispatchers, ensures that the problem of "synchronization-induced overload" statistically will not occur.

This distribution is precomputed once by a server and distributed to all dispatchers. The details of this computation will be described shortly in Section 4.2. We will show that q, the number of $m \times n$ k-matching matrices that form the support of Φ, is at most $(m+n-k)^2$. Hence the distribution can be encoded in a string of $O(q) = O\big((m+n-k)^2\big)$ in length, which is quite short given the parameter settings we are working with right now ($m = 60, n = 20, k = 18$); Broadcasting such a short string to all dispatchers once a day or once an hour clearly incurs only negligible network and system overheads.

Algorithm 1 Weighted Randomized Assignment

Upon the arrival of a new extent
  $\Phi \leftarrow \Phi_i$ with probability p_i, $i = 1, 2, ..., q$;
  Assign its blocks according to Φ;

Here we provide some intuitions why and how this algorithm works, deferring the rigorous reasoning to Section 4.2. With this algorithm executed by the dispatchers for a total of T times, with overwhelming probabilities, approximately p_1T, p_2T, $\cdots$, p_qT assignments are made using $m \times n$ k-matching matrices Φ_1, Φ_2, ..., Φ_q, respectively. As we will show in Section 4.2, the assignment distribution – consisting of the set of $m \times n$ k-matching matrices and their respective probabilities – and T are determined so that if exactly p_iT assignments are made using $m \times n$ k-matching matrix Φ_i, for $i = 1, 2, ..., q$, then the occupant ratio of every cell will

be equal to the same target occupancy $\hat{l}$. Therefore, the occupancy ratios of all cells, resulting from the independent blocks-to-cells assignments made by dispatchers, are close to the target load $\hat{l}$. Conceivably, given the current occupancy ratios of cells, there is a mathematical limit on how low this target ratio $\hat{l}$ can be. As we will show in Section 4.2, our algorithm is optimal in the sense $\hat{l}$ matches this lower bound.

The total number of assignments T the dispatchers collectively need to make in order to restore the balance is clearly determined by this target ratio $\hat{l}$. It however takes some effort for the dispatchers, who do not communicate or coordinate with one another, to collectively hit this target number T, because new file blocks may arrive at dispatchers at different rates. Our solution is very simple. We typically know the number of dispatchers Z within a certain accuracy range (say ± 5). At the time of each adjustment, we simply ask each dispatcher to make T/Z assignments that way. Unless the occupancy ratios are severely unbalanced, this number T/Z is typically much smaller than the number of file blocks a dispatcher needs to add to the data center during two consecutive adjustments. Once a dispatcher reaches the quota T/Z, it is done with the first stage, and enters the second stage of making uniform random assignments. Since all or the vast majority of dispatchers will reach the quota T/Z, their total will be equal to only slightly smaller than T. Even if the latter happens, it has a negligible effect on the efficacy of the load balance, and this "deficiency" can be made up during the next adjustment cycle.

3.3.2 The Cruising Stage

As explained earlier, each dispatcher, after adding T/Z file blocks to the data center using Algorithm 2, will enter the cruising stage. In the cruising stage, a dispatcher adds statistically the same number of blocks to each cell so as not to change the status quo. We achieve it by using a "plane sweep" algorithm at every dispatcher. This algorithm limits the maximum such variation caused by a single dispatcher to at most two blocks. In addition, the algorithm instances at different dispatchers are independently randomly parameterized to keep the actions of different dispatchers, and hence the cells where such maximum variations occur, statistically independent of one another.

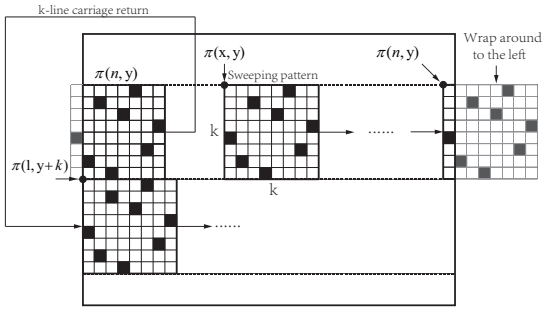

Figure 2. Sweeping algorithm

Algorithm 2 Sweeping algorithm

Upon the arrival of a new extent
  Assign its blocks using $\pi_{(x,y)}$;
  $x \leftarrow x+1 \bmod m$;
  if x = 1 **then**
    $y \leftarrow y+k \bmod n$;
  end if

The basic idea of the algorithm is for a dispatcher to make successive assignments using the same $k \times k$ k-matching matrix π that is shifted in a "sweeping pattern" over time as follows. The sweeping starts with placing π at an initial position (x,y), and making the first blocks-to-cells assignment according to $\pi_{(x,y)}$. Here $\pi_{(x,y)}$ is π with offset (x,y), the semantics of which will be described shortly. The dispatcher, upon the arrival of subsequent extents, sweeps the assignment matrix right horizontally and then "k lines downward" vertically as follows. The k-matching matrices $\pi_{(x+1,y)}$, $\pi_{(x+2,y)}$, and so on will be used to make subsequent assignments. In other words, the assignment matrix is right-shifted by 1 with each assignment. This right-shifting continues until the assignment matrix $\pi_{(n,y)}$ is used. At this point, a "k-line carriage return" happens, and the sweeping continues "k rows down" at the starting position $(1, y+k \bmod m)$, in subsequent assignments. This sweeping pattern, including the right-shifting and the k-line carriage return, is illustrated in Figure 2.

We now precisely specify the semantics of $\pi_{(x,y)}$, the k-matching matrix π with offset (x,y). With $\pi_{(x,y)}$, blocks of a new extent will be assigned to k cells, out of the k^2 cells that lie between the x_{th} and the $(x+k-1 \bmod n)_{th}$ columns and between the y_{th} row and the $(y+k-1 \bmod m)_{th}$ row, according to π as follows. Suppose the k-matching matrix π is encoded as $(a_1, a_2, ..., a_k)$ in its permutation form, which means the matrix elements take value 1 at coordinates $(1,a_1)$, $(2,a_2)$, ..., (k,a_k), and value 0 everywhere else. Then the blocks will be assigned to cells $(x \bmod n, y+a_1-1 \bmod m)$, $(x+1 \bmod n, y+a_2-1 \bmod m)$, ..., $(x+k-1 \bmod n, y+a_k-1 \bmod m)$ respectively. For example, the sweeping pattern used in Figure 2, is an 8×8 8-matching matrix that is equal to $(6,3,5,8,1,7,2,4)$ in its permutation form. We use a $k \times k$ k-matching matrix instead of a $m \times n$ k-matching matrix, for the sweeping operation, because the former results in a more even sweeping of the cloud matrix, as will be explained immediately.

The purpose of this sweeping is to assign almost the same number of blocks to each and every cell over time at a single dispatcher. Indeed, if k divides m, then each cell will receive exactly 1 block during each complete sweep of the entire cloud matrix. Even when k does not divide m however, it can be shown that the numbers of blocks from each dispatcher that any two cells receive respectively at any time will differ by at most 2. Note this difference may increase to as large as

$k-1$ if an $m \times n$ k-matching matrix is used instead of $k \times k$ k-matching matrix, for the sweeping operation.

However, if all dispatchers use the same $k \times k$ k-matching matrix π or start sweeping at the same position (say $(1,1)$), the problem of overloading due to synchronization could happen. To avoid this situation, each dispatcher will generate a uniform random k-matching matrix (i.e., sampled uniformly from the set of all $k!$ such k-matching matrices) and a uniform random starting position independent of one another, and use them for its sweeping operation.

4. Computing the Load-Balancing Distribution

Earlier in Section 3.3.1, we have explained that a dispatcher assigns blocks of a new extent according to a k-matching sampled from a set of k-matchings using a precomputed distribution. What this distribution is and how to compute it are the subject of this section. In Section 4.1, we will mathematically characterize this distribution. In Section 4.2, we will describe a standard algorithm for computing it, which converts this computation problem to a classic Birkhoff-Von Neumann decomposition problem [3, 7]. This standard algorithm however has a high complexity of $O\big((m+n-k)^{4.5}\big)$. Although the standard algorithm is fast enough for the current dimensions of the cloud system ($m = 60, n = 20, k = 18$), it would be too slow for a future cloud system that is much larger (say $m = n = 500$). In Section 4.3, we present an improved algorithm that solves for this distribution with only $O\big((m+n)\log(m+n) + (3k)^{4.5}\big)$ computational complexity.

4.1 Mathematical Characterization

We first make a "fluid" assumption only for the purpose of mathematically characterizing this distribution. We assume that the number of assignments the dispatcher makes according to any k-matching matrix does not have to be a nonnegative integer and instead can be any nonnegative real number. This fluid assumption allows linear algebra machineries to be used for the characterization step. The actual assignments however are always discrete (i.e., non-fluid): Exactly k blocks of data will be assigned to k cells on distinct rows and distinct columns in each assignment.

With this "fluid" assumption, this problem can be restated as follows. Given a nonnegative m by n load matrix L, identify a nonnegative $m \times n$ compensating matrix C such that $C + L \equiv \hat{L}$ is a constant matrix, in which all elements have the same load $\hat{l}$; This compensating matrix C needs to be decomposable in the sense it can be written as a linear combination of $m \times n$ k-matching matrices with nonnegative coefficients. Now suppose we identify such a decomposition,

$$C \equiv \sum_{i=1}^{q} \lambda_i \Phi_i \tag{1}$$

such that $L + C \equiv \hat{L}$, and $\lambda_i > 0$ and Φ_i is an $m \times n$ k-matching matrix, for $i = 1, 2, ..., q$. The distribution used for load-balancing by dispatchers is simply the probability distribution on the finite set $\{\Phi_1, \Phi_2, ..., \Phi_q\}$ that assigns probability p_i to Φ_i,

$$\mathbb{P}[\Phi = \Phi_i] = p_i = \frac{\lambda_i}{\sum_{j=1}^{q} \lambda_j} \tag{2}$$

In other words, upon the arrival at a dispatcher of a new file block to be added to the cloud, the dispatcher samples assignment Φ_i with probability p_i.

This simplified problem has been studied in the mathematics literature [19]. Mendelsohn and Dulmage [19] showed that there is a mathematical lower bound on how small $\hat{l}$, the constant in the constant matrix $C + L$ or the target load of the load-balancing scheme, can be. This lower bound $\hat{l}$ can be derived from the load matrix L as follows.

$$\hat{l} = \max\{\frac{\sum_{i=1}^{m}\sum_{j=1}^{n} l_{ij} - k\min_{1\le j\le n}\{\sum_{i=1}^{m} l_{ij}\}}{mn - mk}, \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} l_{ij} - k\min_{1\le i\le m}\{\sum_{j=1}^{n} l_{ij}\}}{mn - kn}, \max_{1\le i\le m, 1\le j\le n}\{l_{ij}\}\} \tag{3}$$

In other words, a minimum load $\hat{l}$ must be reached for the loads of all cells to become identical. Note this minimum load $\hat{l}$ could be larger than the storage capacity of any cell, in which case perfect load-balancing is not mathematically possible. Even in such an extreme case, it can be shown that our randomized algorithm manages to delay the inevitable event of one disk becoming full to the maximum extent possible.

An equivalent way to look at this lower bound (Formula 3) is the following lemma. It will be used in the next section for proving the sufficiency of our matrix compression algorithm.

LEMMA 1. *[19] A nonnegative $m \times n$ matrix C is decomposable (into a linear combination of k-matchings with nonnegative coefficients) if and only if every row sum and column sum of C is smaller than or equal to $\|C\|/k$, where $\|C\| \equiv \sum_{i=1}^{m}\sum_{j=1}^{n} |c_{ij}|$ is the L_1 norm of, or in this case the sum of all elements in, the matrix C.*

4.2 A Standard Algorithm

The algorithmic question of identifying the linear combination to match this lower bound was also addressed by Mendelsohn and Dulmage [19]. Finding such a linear combination can be converted to the problem of performing the Birkhoff-Von Neumann (BVN) decomposition [3, 7] of an $(m+n-k) \times (m+n-k)$ doubly stochastic matrix. A doubly stochastic $N \times N$ matrix is one in which all elements are nonnegative and all row sums and column sums are equal to 1. According to Birkhoff [3] and Chang, Chen and Huang [7], the number of different k-matchings in the decomposition is at most $(m+n-k)^2$ for both methods.

This conversion is however a mixed blessing, since the standard algorithm for this decomposition has very high computational complexity, namely, $O(N^{4.5})$ for an $N \times N$ matrix. A recently proposed algorithm brings this complexity down to $O\big(N^3 \log N\big)$ [16]. In our current cloud system, $m = 60$, $n = 20$, and $k = 18$, so $m+n-k = 62$. For computing this linear combination, we need to perform a BVN decomposition on a 62×62 matrix, which takes only a few seconds using existing algorithms.

4.3 A Faster Algorithmn

While converting the computation of the probability distribution used in our load-balancing scheme to a $(m+n-k) \times (m+n-k)$ BVN decomposition problem works for the current cloud parameters above, future cloud matrices could be much larger in dimensions ($m \times n$), making this approach too slow to implement. We design a deterministic algorithm that can reduce this problem to a $3k \times 3k$ BVN decomposition problem. It does so by compressing any $m \times n$ decomposable matrix C (for arbitrarily large m and n) into a $2k \times 2k$ decomposable matrix $\tilde{C}$. A suitable probability distribution that is used in our load-balancing scheme can be derived from the decomposition of $\tilde{C}$, in a straightforward manner. Since the decomposition of $\tilde{C}$, is a $(2k+2k-k) \times (2k+2k-k)$ BVN decomposition, and it takes only $O\big((m+n)\log(m+n)\big)$ to convert C into $\tilde{C}$, our algorithm reduces the complexity of computing this distribution from $O\big((m+n-k)^3 \log(m+n-k)\big)$ to $O\big((m+n)\log(m+n) + (3k)^3 \log(3k)\big)$. For small k values that are typical for storage erasure codes, the latter complexity can be several orders of magnitude smaller than the former, for very large cloud size ($m \times n$).

4.3.1 Compressing C into $\tilde{C}$

We compress C into $\tilde{C}$ by collapsing one or more rows of C into a row in $\tilde{C}$ and one or more columns of C into a column in $\tilde{C}$. Here we explain only the row collapsing process in details since the column collapsing process is similar to and independent of the row collapsing process. Let α_1, α_2, ..., α_m be the sums of rows 1, 2, ..., m of matrix $C = (c_{ij})$, respectively, i.e., $\alpha_i \equiv \sum_{j=1}^{n} c_{ij}$ for $i = 1, 2, ..., m$. We first partition the index set $\{1, 2, ..., m\}$ of the rows of the matrix C into $2k$ disjoint index sets A_1, A_2, ..., A_{2k}, according to the row sums α_1, α_2, ..., α_m. After the partition, the sum of row sums in each index set A_i, namely $\sum_{j \in A_i} \alpha_j$, is guaranteed to be no more than $\|C\|/k$. This guarantee allows the compressed matrix $\tilde{C}$ to be decomposable, as we will explain shortly.

How to identify such partitions $A_1, A_2, ..., A_{2k}$ is clearly an instance of the classical bin packing problem, in which we would like to pick rows $1, 2, ..., m$ with "weights" α_1, α_2, ..., α_m respectively, into bins A_1, A_2, ..., A_{2k}, so that no bin has a total weight larger than $\|C\|/k$, the aforementioned guarantee. Our algorithm for computing such a partition, shown in Algorithm 3, is the first fit decreasing (FFD) strategy [14] that is among the simplest heuristic algorithms for solving this NP-complete problem. The objects (rows), indexed by the elements of array U, are first sorted in the increasing order according to their weights, and then packed into one of these $2k$ bins one after another in the increasing order. When the algorithm tries to pack an object, it will try bin A_1 first, bin A_2 next, and so on, until it finds a bin that can hold it. By a slight abuse of notation, we use α_{A_i} to denote $\sum_{j \in A_i} \alpha_j$, and use it in line 5 of Algorithm 3. Since the total weight of all rows is $\|C\|$, and each bin holds at most $\|C\|/k$, a minimum of k bins are needed for a bin packing solution to exist. The FFD strategy guarantees, among other nice properties, that no more than twice this minimum, which is $2k$ in this case, is needed [12].

Algorithm 3 Collapsing of Rows by FFD

Input: Decomposable m by n matrix C;
Output: Row index sets $A_1, A_2, ..., A_{2k}$;

1: Let the array $U[1..m]$ store the indices of rows such that $\alpha_{U[1]} \le \alpha_{U[2]} \le ... \le \alpha_{U[m]}$ (i.e., with their sums sorted in the increasing order);
2: $A_1, A_2, ..., A_{2k} \leftarrow \emptyset$; ▷ Initialize to empty sets
3: $j \leftarrow 1$; ▷ Initialize to the first set
4: **for** $i \leftarrow 1$ to m **do**
5: **while** $\alpha_{A_j} + \alpha_{U[i]} > \|C\|/k$ **do**
6: $j \leftarrow j + 1$;
7: **end while**
8: $A_j \leftarrow A_j \bigcup \{U[i]\}$;
9: **end for**

Let β_1, β_2, ..., and β_n be the sums of columns 1, 2, ..., n of matrix C, respectively, i.e., $\beta_j \equiv \sum_{i=1}^{m} c_{ij}$ for $j = 1, 2, ..., n$. We can similarly partition the index set $\{1, 2, ..., n\}$ of the columns of the matrix C into $2k$ disjoint index sets B_1, B_2, ..., B_{2k}, according to the column sums β_1, β_2, ..., β_n, also using Algorithm 3 but with different parameters. Similarly, after the partition, the sum of column sums in each index set B_j, namely $\sum_{i \in B_j} \beta_i$, is guaranteed to be no more than $\|C\|/k$. We will explain shortly that this guarantee, in combination with the guarantee that the sum of row sums in each A_i is upper bounded by $\|C\|/k$, ensures that $\tilde{C}$ is decomposable.

4.3.2 Compute the Distribution

The row partitions A_1, A_2, ..., A_{2k} and column partitions B_1, B_2, ..., B_{2k} dictate how C is compressed into $\tilde{C}$: Rows of C in each A_i collapse into row i of $\tilde{C}$, for $i = 1, 2, ..., 2k$; Columns of C in each B_j collapse into column j of $\tilde{C}$, for $j = 1, 2, ..., 2k$. More specifically $\tilde{C} = (\tilde{c}_{ij})$ and $\tilde{c}_{ij} = \sum_{h \in A_i} \sum_{s \in B_j} c_{hs}$, for $i, j = 1, 2, ..., 2k$. Clearly $\|\tilde{C}\|$, the L_1 norm of $\tilde{C}$, is the same as $\|C\|$, since A_1, A_2, ..., A_{2k} and B_1, B_2, ..., B_{2k} are partitions of the m rows and n columns of C respectively. As explained above, the sum of each row i of $\tilde{C}$, equal to α_{A_i}, is no more than $\|C\|/k = \|\tilde{C}\|/k$; So is the sum of each column j of $\tilde{C}$. Therefore $\tilde{C}$ is decomposable according to Lemma 1 above.

Since $\tilde{C}$ is decomposable, it can be written as a linear combination of $2k \times 2k$ k-matching matrices with nonnegative coefficients, which induces a probability distribution $\tilde{\mu}$ on the set of $2k \times 2k$ k-matching matrices like in Formula 2. The probability distribution μ we use to assign blocks to the m by n cloud matrix is then derived from $\tilde{\mu}$: $\tilde{\mu}(\{\Phi_i\}) \equiv \mathbb{P}[\Phi = \Phi_i]$ as follows. Suppose a $2k \times 2k$ k-matching matrix $\Delta = \{\delta_{ij}\}$ is sampled (with a probability as specified in $\tilde{\mu}$). Suppose the matrix elements δ_{i_1,j_1}, δ_{i_2,j_2}, ..., δ_{i_k,j_k} (on distinct rows and columns) take value 1, and the rest take value 0. Then the first block is assigned to the cell indexed by a (random) coordinate (h_1,s_1) sampled from the set $\Gamma_1 \equiv \{(h,s)|h \in A_{i_1}, s \in B_{j_1}\}$ with the following probability distribution: $\mathbb{P}[(h_1,s_1) = (h,s)|(h,s) \in \Gamma_1] = c_{h,s}/\tilde{c}_{i_1,j_1}$. Note these probability values add up to 1, because $\sum_{(h,s)\in\Gamma_1} c_{h,s} = \tilde{c}_{i_1,j_1}$, due in turn to the fact that we collapse rows A_{i_1} and columns B_{j_1} of C respectively into row i_1 or column j_1 of $\tilde{C}$. The second, the third, ..., and the k_{th} blocks are similarly assigned to cells indexed by random coordinates (h_2,s_2), (h_3,s_3), ..., (h_k,s_k) respectively. By noting that the (random) coordinates (h_1,s_1), (h_2,s_2), ..., (h_k,s_k) are mutually independent, we have completed the specification of μ.

5. Evaluation

In this section, we evaluate through simulations the efficacy of our load-balancing scheme. We first describe in Section 5.1 the common settings, parameters, and metric in all simulation studies, and then report their findings in subsequent sections.

5.1 Common Settings and Parameters

Unless otherwise stated, in simulation studies, we use the ballpark system parameters provided to us by the top-tier cloud service provider we are working with. The cloud system of the provider can be modeled as an $(m = 60) \times (n = 20)$ matrix of cells. Each cell contains 10 storage servers (about a half rack), and each storage server has 40 disks of 8TB capacity each. As explained earlier, we assume each (file) extent consists of $k = 14 + 4 = 18$ coded blocks and each block has size $3\text{GB}/14 \approx 214\text{MB}$. Hence each cell can hold $10 \times 40 \times 8\text{TB}/(214\text{MB}) \approx 1.5 \times 10^7$ blocks. We set the total arrival rate of new extents (or blocks) to the cloud system to 0.1% of the (current) total data center storage capacity per day. We also assume that there are 5,000 dispatchers in the system and all dispatchers obtain a daily report of the load occupancy ratios of all cells and will adjust the load distribution used in the load-balancing scheme accordingly. While this once-a-day (load) reporting frequency may sound too low to some readers, we emphasize that the performance of our load-balancing scheme can only improve with higher reporting frequencies.

In all simulation studies, we use metric $D = (\max l_{ij} - \bar{l})/v$, the difference between the load ratio of the most heavily loaded cell and the average load ratio $\bar{l} = \frac{\|L\|}{mn} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} l_{ij}}{mn}$, to measure how well loads are balanced across the cells, and refer to it as the *positive load ratio deviation*. We would like this metric D to be as small as possible, so that $1 - D$, the maximum reachable utilization ratio of the data center, can be as close to 1 (i.e., 100%) as possible.

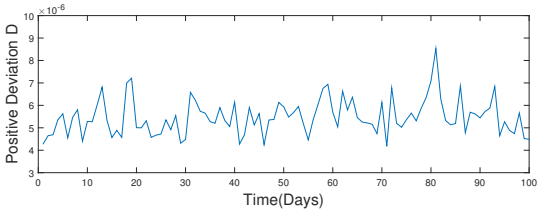

Figure 3. The fluctuation of D over time.

5.2 Near-Perfect Load Balance

In this section, we study, when new file blocks are added to a data center, how evenly our load-balancing scheme can distribute the load across the cells. We assume the loads across all cells are perfectly balanced to start with, and the utilization ratio of each cell is 70%. Recall that 5,000 dispatchers collectively add new blocks at a rate of 0.1% of the total data center storage capacity per day. We have performed 1200 simulation runs, each of which simulates the process, over a period of 100 days, of filling up the data center to 80% utilization. We have found that the positive load ratio deviation D exceeds 0.000675% in only 1% of the simulation runs. In other words, the utilization ratio of the data center can reach 79.999325% with probability 0.99, without any cell's load ratio exceeding 80%. This result demonstrates that our scheme keeps the storage loads across the cells near-perfectly balanced, thus allows the data center to use almost all its storage capacity if needed.

We have also taken a closer look at how the positive load ratio deviation D fluctuates over the 100-day period. We run the previous simulation once more using the same parameters. The resulting "sample path" is shown in Figure 3. We can see from Figure 3 that D stays within a narrow range between 0.000417% and 0.000854%. This indicates that our load-balancing scheme maintains a near-perfect balance of storage loads across cells during the entire 100-day period, when new (file) blocks arrive continually.

5.3 Comparison with Uniform Random Assignment

In this section, we compare the performance of our load-balancing scheme, consisting of the weighted randomized assignment (Section 3.3.1) and the sweeping (Section 3.3.2) algorithms, with that of uniform random assignment. We compare it also with that of continuing to run the weighted randomized assignment algorithm, instead of the sweeping algorithm, even during the cruising stage. We assume that each cell has an initial load independently uniformly randomly distributed in the interval $[50\%, 51\%]$. We conduct

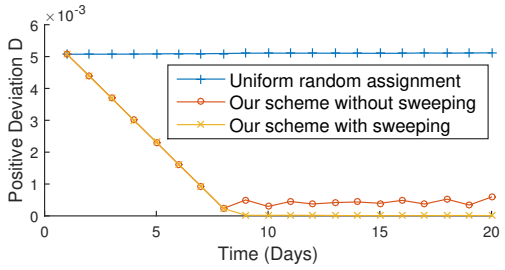

Figure 4. Comparison with uniform random assignment.

one simulation run for all three cases (with sweeping, without sweeping, uniform random) using the same arrival instance (i.e., with the same *outcome* ω) of new file blocks. The simulation results are shown in Figure 4.

We can see that the D curve of the uniform random assignment is flat over time. This means that, starting with an uneven initial load distribution, the uniform random assignment algorithm cannot make it any more balanced over time, which is to be expected because it has no load-balancing power. In comparison, with the other two load-balancing schemes (with sweeping, without sweeping), the D value decreases rapidly until it reaches a small value (around 0.0234%) during the common load-balancing stage. At this point, our scheme with sweeping will enter the cruising stage (i.e., starting to "sweep"), while that without sweeping will continue to balance the load using the weighted randomized assignment algorithm. It is clear from Figure 4 that loads across cells are much more evenly balanced with sweeping than without.

5.4 Two Breakdown Scenarios

We have explained earlier that a hardware failure or a software update could take a row (hardware failure zone) or a column (software update zone) offline for as long as a week (7 days). In the meantime, all new blocks are assigned to other rows or columns not affected by the breakdown, leaving the affected row or column underloaded. In this section, we show that our weighted randomized assignment algorithm can promptly restore the load-balancing after the affected row or column is back online.

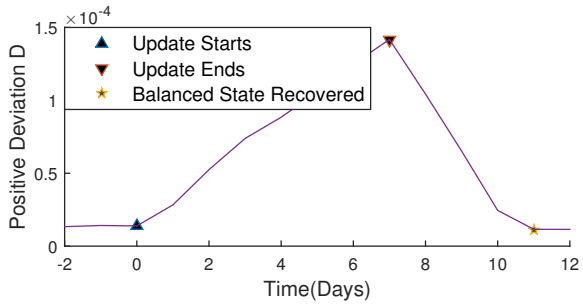

Figure 5. The evolvement of D when a hardware failure happens.

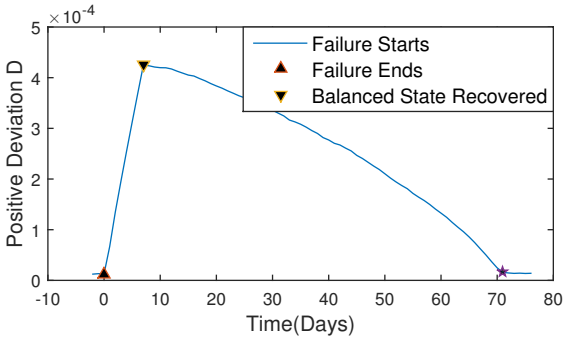

Figure 6. The evolvement of D when a software update happens.

We consider two breakdown scenarios: A hardware failure that takes out a row (containing 20 cells), and a software update that takes out a column (containing 60 cells), both for seven days (from the 0th day to the 7th day in both figures). Note that a software update takes out 3 times as many cells as a hardware failure. We perform 100 simulation runs for each scenario. Figure 5 shows the simulation results for the hardware failure scenario. From the plot, we can see that during the breakdown period, D increases to about 0.003% of cell capacity. After the row comes back online, our weighted randomized assignment algorithm restores the balance within four days, or after new blocks, in the amount no more than $0.1\% \times 4 = 0.4\%$ of the total storage of the cloud system, are assigned.

The simulation results for the software update scenario are shown in Figure 6. Software update takes much more time (about 64 days) to recover than the hardware failure, because the former takes out three times as many cells as the latter. In fact, even with the idealized centralized load-balancing scheme, it takes at least 63 days to restore the load balance, as shown in the following analysis, so our scheme is near-optimal in this regard.

When a column is taken out, the other 19 columns each receives about $1/19$ of the incoming storage load over the next seven days. Consequently, when the updating column comes back to service (i.e., restored) after seven days, it has on average $\frac{7}{19}w$ less load than others, where w is the number of new blocks that arrive each day. Since the restored column can take in at most $\frac{1}{18}w$ new blocks each day afterwards, and any other column on average takes in at least $\frac{1-1/18}{19}w = \frac{17}{342}w$ new blocks each day, the restored column can take in at most $(\frac{1}{18} - \frac{17}{342})w = \frac{1}{171}w$ more new blocks each day than any other column on average. Hence it takes at least $(\frac{7}{19}w)/(\frac{1}{171}w) = 63$ days for the restored column to make up the load "deficit".

Note although it takes quite a long time for the load ratio of the restored column to "catch up" with that of others, the resulting decrease in the maximum reachable utilization ratio of the data center, even at its largest (immediately after the column is restored), is minute. In fact, from Figure 6, we

can see that the peak value of D during a software update is only about 0.05%, which means the maximum achievable utilization ratio can still reach 79.95% without any cell's load ratio exceeding 80%. However, if we really would like to shorten this "recovery time", we can achieve that by increasing the number of software update zones. For example, if we had 30 instead of 20 software update zones, the recovery time would decrease to 12 days according to our simulations.

6. Related Work

The problem of balancing storage loads in a cloud or data center has been studied, mostly in the context of achieving fault-tolerance through data replication, in works such as Google file system [15], Hadoop distributed file system [5], DepSky storage system [2] and Ceph file system [26]. Chen et. al [9] proposed a proactive Markov Decision Process-based load-balancing algorithm, which efficiently migrates load from heavy-loaded VMs to light-loaded ones. Wei et. al [25] proposed a model to capture the tradeoff between availability and level of replication.

Achieving fault-tolerance via space-efficient erasure coding has also attracted much research attention. Zhang et. al [27] compared the performance (in achieving fault-tolerance) of erasure coding and that of replication in Hadoop [5]. Dimakis et. al [13] and Sathiamoorthy et. al [23] focused on the data recovery issue in erasure-coded storage system. Dimakis et. al [13] proposed a information-theoretic framework that can be used to determine the minimum amount of information that has to be communicated in order to repair failures. Sathiamoorthy et. al [23] presented a novel family of erasure codes that offer better coding efficiency and higher reliability than Reed-Solomon codes [6, 18]. None of these works, however, has considered anything close to the aforementioned 2D combinatorial constraint (i.e., the diversity requirement) that we have to work under in solving our load-balancing problem.

Interestingly, load balancing under constraints similar to ours has been studied in the contexts of network switching and CPU scheduling in parallel and cluster computing. Some of the early works on cluster computing have examined load balancing issues that are slightly similar to ours. Work by Rudolph et. al [22] studies CPU load-balancing in a parallel computer. Its core idea is that, when a processor is ready to schedule the next job in its local job queue, it will ask – with a probability that is a function of the queue length – another processor, chosen uniformly randomly from the set of other processors, about its queue length. If the queue length of that processor is much shorter, the job will be transfered there. Chow et. al [11] analyze the performance of four scheduling policies in single-dispatcher distributed systems. Our problem setting is very different than that in this work, as there is no communication among dispatchers and the blocks have to be assigned according to a k-matching matrix. In other load balancing schemes designed for multi-server or multi-processor systems, either a central server is used for all task assignments [4, 11] or the job dispatchers are able to communicate with one another [1, 10].

Chang et. al's work on crossbar switching [7, 8] is similar to ours in its solution approach and the use of BVN decomposition.. Both schemes employ a two-stage process, wherein randomized assignments are performed in the first stage to reach load balance, and a "sweeping" style algorithm is used in the second stage to maintain the balance. There are two key differences however. First, their work deals only with square matrices (i.e., $N \times N$) and full-size matchings (i.e., N-matching) while our work has to deal with any matrix shape and matching size. This makes our problem more mathematically complicated. Second, in a switching setting, once a matrix is BVN-decomposed into a linear combination of matching matrices, input ports can effectively coordinate with one another to translate the decomposition result into a deterministic sequence of matchings to be carried out in succession. Our work, on the other hand, has to harness the randomness caused by the distributed independent dispatching requirement.

Work by Porter et. al [21] studies how to efficiently route network traffic inside a data center by viewing it as a "giant switch." In this work, like in [7, 8], the BVN decomposition of the "switch"-wide traffic matrix (i.e., the demand) is performed to obtain a sequence of configurations (i.e., "matchings") that should be used over time to meet the demand. Besides the use of BVN decomposition, there is no other close similarity between our work and theirs. In addition, the aforementioned two primary differences between our work and [7, 8] also apply in this case.

7. Conclusion

In this work, we have proposed a distributed randomized solution to the problem of balancing the storage loads of disks in a large cloud system, under the requirements of no communication or coordination among storage load dispatchers and placing blocks of a file on distinct rows and columns of the cloud matrix. We have also designed an efficient algorithm for computing the probability distribution used in our solution, which allows the solution to be applicable to much larger cloud systems. Our simulations using ballpark parameters provided by a tier-1 cloud service provider show, among other things, that our solution can balance heterogeneous load very well in theoretically minimum time and keep this balanced state stable, in the most challenging operation scenarios. In the future, we will study how to best achieve and maintain load-balancing across the data center, when deleting an old row of cells (i.e., retiring racks of old servers) or adding a new row, under the same operational constraints (e.g., the diversity requirement).

Acknowledgments

We thank anonymous reviewers for their insightful comments and suggestions that help improve the quality of the paper. We also thank Mr. Jian Huang and Prof. Richard Peng for their feedbacks on earlier drafts of this paper. This research is supported in part by NSF awards CNS-1302197 and CNS-1423182.

References

[1] K. Baumgartner and B. Wah. Gammon: a load balancing strategy for local computer systems with multiaccess networks. *Computers, IEEE Transactions on*, 38(8):1098–1109, Aug 1989. ISSN 0018-9340. .

[2] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[3] D. Birkhoff. Tres observaciones sobre el algebra lineal. *Universidad Nacional de Tucuman Revista , Serie A*, 5:147–151, 1946.

[4] F. Bonomi and A. Kumar. Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler. *Computers, IEEE Transactions on*, 39(10):1232–1250, Oct 1990. ISSN 0018-9340. .

[5] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[6] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[7] C.-S. Chang, W.-J. Chen, and H.-Y. Huang. Birkhoff-von neumann input buffered crossbar switches. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1614–1623 vol.3, Mar 2000. .

[8] C.-S. Chang, D.-S. Lee, and Y. shean Jou. Load balanced Birkhoff-von Neumann switches, Part I: One-stage Buffering, 2001.

[9] L. Chen, H. Shen, and K. Sapra. Distributed autonomous virtual resource management in datacenters using finite-markov decision process. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[10] Y.-C. Chow and W. H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *Computers, IEEE Transactions on*, C-28(5):354–361, May 1979. ISSN 0018-9340. .

[11] Y.-C. Chow and W. H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *Computers, IEEE Transactions on*, 100(5):354–361, 1979.

[12] F. Clautiaux, M. DellAmico, M. Iori, and A. Khanafer. Lower and upper bounds for the bin packing problem with fragile objects. *Discrete Applied Mathematics*, 163, Part 1(0):73 – 86, 2014. ISSN 0166-218X. . Matheuristics 2010.

[13] A. G. Dimakis, P. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *Information Theory, IEEE Transactions on*, 56(9): 4539–4551, 2010.

[14] G. Dósa. The tight bound of first fit decreasing bin-packing algorithm is $FFD(I) \leq 11/9OPT(I) + 6/9$. In *Proceedings of the First International Conference on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, ESCAPE'07, pages 1–11, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74449-5, 978-3-540-74449-8. URL dx.doi.org/10.1007/978-3-540-74450-4_1.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[16] A. Goel, M. Kapralov, and S. Khanna. Perfect matchings in $O(n \log n)$ time in regular bipartite graphs. *CoRR*, abs/0909.3346, 2009. URL arxiv.org/abs/0909.3346.

[17] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, Boston, MA, 2012. USENIX. ISBN 978-931971-93-5. URL bit.ly/1Bqj7CI.

[18] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *FAST*, page 20, 2012.

[19] N. S. Mendelsohn and A. L. Dulmage. The convex hull of sub-permutation matrices. *Proceedings of the American Mathematical Society 9*, pages 253–254, 1958.

[20] Microsoft Azure Storage Team. Introducing zone redundant storage. bit.ly/1LFsA44, 2014.

[21] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 447–458, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. . URL doi.acm.org/10.1145/2486001.2486007.

[22] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 237–245, New York, NY, USA, 1991. ACM. ISBN 0-89791-438-4. . URL doi.acm.org/10.1145/113379.113401.

[23] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[25] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng. Cdrm: A cost-effective dynamic replication management scheme

for cloud storage cluster. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 188–196. IEEE, 2010.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[27] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010*, 52, 2010.