Deterministic Data Distribution for Efficient Recovery in Erasure-Coded Storage Systems

Liangliang Xu, Min Lyu, Zhipeng Li, Yongkun Li, and Yinlong Xu

Abstract—Due to individual unreliable commodity components, failures are common in large-scale distributed storage systems. Erasure codes are widely deployed in practical storage systems to provide fault tolerance with low storage overhead. However, random data distribution (RDD), commonly used in erasure-coded storage systems, induces heavy cross-rack traffic, load imbalance, and random access, which adversely affects failure recovery. In this article, with orthogonal arrays, we define a Deterministic Data Distribution (D^3) to uniformly distribute data/parity blocks among nodes, and propose an efficient failure recovery approach based on D^3, which minimizes the cross-rack repair traffic against a single node failure. Thanks to the uniformity of D^3, the proposed recovery approach balances the repair traffic not only among nodes within a rack but also among racks. We implement D^3 over Reed-Solomon codes and Locally Repairable Codes in Hadoop Distributed File System (HDFS) with a cluster of 28 machines. Compared with RDD, our experiments show that D^3 significantly speeds up the failure recovery up to 2.49 times for RS codes and 1.38 times for LRCs. Moreover, D^3 supports front-end applications better than RDD in both of normal and recovery states.

Index Terms—Distributed storage system, erasure coding, traffic, orthogonal array, load balance

1 Introduction

A large-scale Distributed Storage System (DSS) typically consists of many individual unreliable commodity components, which induces frequent component failures [1], [2]. In order to guarantee high reliability and availability against component failures, a common approach is to store data with redundancy. *Replication* and *erasure coding* are two common approaches to provide fault tolerance.

Replication provides the simplest form of redundancy by storing multiple copies of data. For example, Google File System (GFS) [3] and Hadoop Distributed File System (HDFS) [4] store three copies of each data block by default to tolerate double component failures. Replication is easy to deploy and recover against failures. However, the storage overhead is too high, say 3× for triplicate.

As an alternative, erasure codes achieve the same fault tolerance as replication with much lower storage overhead and are commonly deployed in modern DSSes [2], [5], [6], [7]. For example, Google's ColossusFS (the successor to GFS) and Facebook's HDFS-RAID [7] deploy Reed-Solomon (RS) codes [8], which reduce storage redundancy to 1.5× and 1.4× respectively, compared to 3× in traditional triplicate. But with erasure codes, recovering a failed block needs to retrieve multiple available blocks, which induces high repair cost. For example, in a Facebook's DSS storing multiple petabytes of RS coded data, a median of 180 terabytes of repair traffic is generated per day [2]. Though erasure codes improve storage efficiency, they significantly increase disk and network traffic for failure recovery.

Compared to RS codes, Locally Repairable Codes (LRCs) are a class of efficiently repairable codes and offer higher reliability with additional local parity blocks [5], [9], [10], by which LRCs significantly reduce occupation of bandwidth in failure recovery. Many IT infrastructure companies begin to consider deploying wide stripes to provide competitive reliability, low storage overhead and high levels of resiliency. In such a scene [11], one file divided into many data blocks, say 150, and encoded to a small number of parity blocks, say 4, the reconstruction bandwidth cost of RS coded stripes is insufferable, while LRCs can save lots of network bandwidth, and thus becomes a good choice for the deployment of wide stripes.

Fast recovery is one key goal for DSSes to avoid secondary failures or irreparable failures. In addition, the reconstruction tasks have to compete with online applications [12], [13], e.g., MapReduce tasks and data queries, for system resources, so in failure recovery, low interference on front-end users' requests is another key goal. Achieving the two goals concurrently in DSSes is interesting and challenging.

A large-scale DSS is deployed over a number of physically independent storage nodes, which are organized into multiple racks, each rack consisting of multiple nodes, and usually adopts random block placement [2], [6], [9], [14], [15] to achieve load balance among different nodes and different racks. To maximize the data availability in DSSes deployed with erasure codes, different blocks of an erasure-coded stripe are stored in nodes of different racks, one block per rack [1], [5], [6], [9], [16]. This "one block per rack" block placement makes a system tolerate the same number of node failures and rack failures.

• *The authors are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China. E-mail: {llxu, lizhip}@mail.ustc.edu.cn, {lvmin05, ykli, ylxu}@ustc.edu.cn.*

Manuscript received 3 Dec. 2019; revised 5 Apr. 2020; accepted 7 Apr. 2020. Date of publication 16 Apr. 2020; date of current version 8 May 2020. (Corresponding author: Min Lyu.) Recommended for acceptance by J. Wang PhD. Digital Object Identifier no. 10.1109/TPDS.2020.2987837

However, random block placement degrades failure recovery due to two reasons. (1) It can not achieve reconstruction read/write load balance well within short ranges of successive stripes. Such "local load balancing" is crucial for repair performance [17]. (2) Failure recovery based on random block placement induces heavy random access load to retrieve surviving blocks, which will slow down the recovery process. Furthermore, the "one block per rack" block placement inevitably makes the repair of any failed block retrieve available blocks from other racks, and as a result, triggers a large amount of cross-rack traffic. In DSSes, the inner-rack bandwidth is considered to be sufficient, but the cross-rack bandwidth is constrained. Typically, the available cross-rack bandwidth per node is only 1/20 to 1/5 of the inner-rack bandwidth [18], [19]. Thus, cross-rack bandwidth is often considered to be a scarce resource [20], [21]. Too heavy cross-rack traffic unavoidably delays the recovery process.

In this paper, we propose a Deterministic Data Distribution (D^3) scheme for the data placement in erasure-coded DSSes. D^3 relaxes the constraint of "one block per rack" and places multiple blocks of the same stripe to different nodes within the same rack to reduce the cross-rack traffic for single-node failure recovery. Because in DSSes, more than 90 percent of component failures are single-node failures, while double rack failures rarely happen [1], [6], [22], D^3 still reaches high reliability. D^3 also achieves "local load balancing" by deterministically distributing data and parity blocks to significantly speed up the failure recovery. Thanks to orthogonal arrays, D^3 ensures read/write load balance even when the storage systems are in different states, such as normal mode and failure recovery, and thus supports front-end applications better and has low interference on the users' requests in recovery. In summary, our contributions can be summarized as follows.

- We design the data layout of D^3 via orthogonal arrays to deterministically distribute blocks to all nodes in the storage systems, and theoretically prove that D^3 provides uniform data layout for RS codes and LRCs in normal state.
- Based on D^3, we present a recovery algorithm which minimizes the cross-rack repair traffic and achieves load balance of recovery traffic against single node failure. So D^3 further guarantees high-quality services for front-end applications.
- We implement D^3 in *HDFS Erasure Coding* [16] shipped with HDFS in Apache Hadoop 3.1.x, and conduct experiments with a cluster of 28 machines with different architectures to evaluate its performance. Results show that D^3 speeds up the failure recovery process up to 2.49 times for RS codes and 1.38 times for LRCs compared with random data distribution.

The rest of this paper is organized as follows. We present some background in Section 2. We motivate this work and present the main idea of D^3 via an example in Section 3. The data layout of D^3 and the single-node failure recovery algorithm are presented in Sections 4 and 5 respectively. The experimental results are presented in Section 6. Section 7 reviews the related works, and finally Section 8 concludes this paper. In order to follow the main logic flow of this paper more easily, we put the proofs of the lemmas and theorems in the technical report [23].

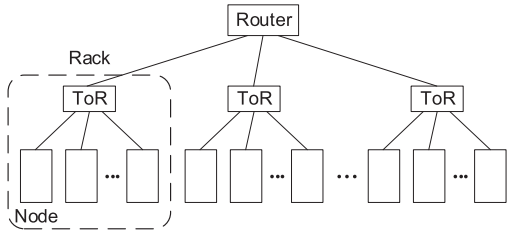


Fig. 1. The typical architecture of distributed storage systems.

2 Background

In this section, we first introduce the typical architecture of DSSes, then review two popular categories of erasure codes, Reed-Solomon (RS) codes [8] and Local repairable codes ($LRCs$) [5], [9], [10], and finally introduce orthogonal arrays [24], the foundation to define D^3.

2.1 Distributed Storage System Architecture

As shown in Fig. 1, a typical distributed storage system (DSS), such as GFS [3], HDFS [4], Azure [5], and Ceph [14], consists of some racks, each rack containing multiple nodes (or storage servers). All nodes within the same rack are connected by a top-of-rack (ToR) switch, while all racks are connected by a central router. The available cross-rack bandwidth suffers from severe competition because many read or write operations (e.g., degraded read, shuffle/join traffic of computing jobs) access data across different racks [20]. As a result, cross-rack bandwidth is considered to be a more scarce resource than inner-rack bandwidth. So we think that cross-rack data traffic is crucial to fast failure recovery in DSSes. To store and retrieve a large amount of data, most DSSes use append-only write and store files as a collection of *blocks* of fixed-size, which form the basic read/write data units. Such DSSes include GFS [3], HDFS [4], and Azure [5].

2.2 Reed-Solomon Codes

Given two positive integers, k and m, a (k, m) code encodes k data blocks into m additional *parity* blocks, such that any one of the $k + m$ blocks can be reconstructed from any other k ones. All the $k + m$ data/parity blocks form a *stripe* and $k + m$ is called *stripe size*. Thus, a (k, m) code tolerates any m blocks being lost in a stripe, and it achieves the so-called *maximum distance separable* (MDS) property [25]. Note that our D^3 can be used to any MDS codes. Since RS codes [8] are the most popular MDS codes that are widely deployed in real DSSes [1], [2], [7], we present D^3 over RS codes and compare its performance with traditional random distribution of blocks in this paper. Furthermore, RS codes satisfy the property of *linearity* [25]. That is, given a (k, m)-RS code, any block B' is the linear combination of any other k different blocks $B_0, B_1, \ldots, B_{k-1}$ in the same stripe. We can get B' by $B' = \sum_{i=0}^{k-1} c_i B_i$, where the c_i's $(0 \leq i \leq k-1)$ are the decoding coefficients specified by the given RS code. Note that additions and multiplications here are performed in a finite field.

2.3 Locally Repairable Codes

Local repairable codes ($LRCs$) are a representative family of non-MDS codes deployed in Microsoft Azure and Facebook [5], [9]. For maximum rack-level fault tolerance, each block of one LRC stripe is usually placed in a different rack. A (k, l, g)-LRC [5], [9], [10] stripe divides k data blocks into l local groups, and maintains a parity block (called a *local parity block*) for each local group, and meanwhile derives g parity blocks (called *global parity blocks*) from the k data blocks similarly with MDS codes. Refer to Fig. 6 for an example. In general, LRCs have the following properties:

- Arbitrary $g+1$ node failures can be recovered, and up to $g+l$ failures can also be recovered if they are information-theoretically decodable [5].
- Single data block/local parity block can be reconstructed by $\frac{k}{l}$ blocks within its local group.
- Global parity block can be reconstructed by other parity blocks.

LRC brings competitive properties, and introduces more types of blocks and different reconstruction flows, which will bring great challenges in designs of data layout and recovery algorithm to achieve load balance in both of normal and recovery states.

2.4 Orthogonal Array

Orthogonal array [24] is a class of combinatorial designs, which are widely used in the design of experimental setting, coding theory, cryptography, and software testing. In particular, orthogonal arrays are used to define the data layout of D^3 in this paper.

Definition 1 [26]. *An orthogonal array $OA(n, k)$ is an $n^2 \times k$ array, $\mathcal{A}$, with entries from a set X of cardinality n such that, within any two columns of $\mathcal{A}$, every ordered pair of symbols from X occurs in exactly one row of $\mathcal{A}$.*

From Definition 1, we can easily obtain the following properties of orthogonal arrays, which guarantee the uniform data distribution for D^3, either for normal or single-failure recovery accesses.

Property 1. *Each column of array $OA(n, k)$ contains each of the n entries equally often (say n times).*

Property 2. *Given $x \in X$ in the ith column, each of the $n-1$ pairs (x, y) appears equally often in the ith and jth columns (say once), where $y \in X - \{x\}$.*

For example, Fig. 5d shows an orthogonal array $\mathrm{OA}(5, 4)$ of the entry set $X = \{0, 1, 2, 3, 4\}$, which is a 25×4 matrix. Each entry of X appears 5 times in any column and each ordered pair (i, j) with $i, j \in X$ appears exactly once in any two columns. Specifically, for each entry, say 0, in the 3rd column, the pair consisting of 0 and every other entry in the 4th column appears exactly once, marked with yellow rectangles.

Theorem 1 [26]. *Let $n = p_1^{e_1} \dots p_v^{e_v}$ be the factorization of n into primes and let $k = \min\{p_i^{e_i} + 1 : i = 1, \dots, v\}$. Then there exists an $OA(n, k)$.*

Given an integer n, Theorem 1 tells an integer k for the existence of an $\mathrm{OA}(n, k)$. The construction for such an $\mathrm{OA}(n, k)$ is proposed in [24]. With such construction, there are at least $k-1$ columns being identical in the first n rows, which will be used in the design of D^3. For example, Fig. 5d shows an OA(5,4), where all columns are identical in the first five rows.

TABLE 1
Major Notations Used in This Paper

Notation	Description
r	Number of racks in the system
n	Number of nodes in a rack
R_i	The ith rack in the system
$N_{i,j}$	The jth node in the ith rack
k	Number of data blocks in a stripe
m	Number of parity blocks in a stripe
len	Stripe size
N_g	Number of groups within a stripe, $N_g = \lceil len/m \rceil$
S_i	The ith stripe
$d_{i,j}$	The jth data block in stripe S_i
$p_{i,j}$	The jth parity block in stripe S_i
$\mathcal{A}$	An $\mathrm{OA}(n, N_g)$ for load balance within a rack
$\mathcal{A}'$	An $\mathrm{OA}(r, N_g + 1)$ for load balance in all racks
G_j^i	The jth region-group of the ith stripe region
G_j^{i*}	G_j^i with recovered blocks
H_i^j	New region-group with recovered blocks of the ith stripe region
l	Number of local parity blocks in an LRC stripe
g	Number of global parity blocks in an LRC stripe

3 Motivations and an Example

In this section, we first present our goals for deterministically distributing blocks in DSSes based on erasure codes to motivate our work, and then explain the main idea of D^3 via an example. We summarize the major notations used in this paper in Table 1.

3.1 Motivations

To maximize the reliability of DSSes based on erasure codes, blocks are usually distributed such that each rack contains at most one block of the same stripe. Such block placement allows a system to tolerate the same number of node failures and rack failures. Because each rack contains at most one block of the same stripe, failure recovery will induce heavy cross-rack traffic, which prolongs the recovery time. However, rack failures are much rarer than node failures in practical systems, and cross-rack bandwidth is often considered to be a more scarce resource than inner-rack bandwidth. So it is viable to tolerate fewer rack failures than node failures so as to reduce cross-rack repair traffic. Such as in HDFS and GFS with three replicas for fault tolerance, three copies of a block are stored in three nodes, where two nodes come from the same rack, while the other one is in another rack. So HDFS protects against either double node failures or a single rack failure. We can distribute blocks of an erasure-coded stripe similarly to HDFS to reduce cross-rack traffic for failure recovery while keeping high reliability.

Furthermore, DSSes usually distribute blocks to randomly chosen nodes and racks for load balance. Specifically, in a system deploying a (k, m)-RS code or (k, l, g)-LRC, the blocks of the same stripe are stored on different randomly chosen nodes, each node in a separate randomly chosen

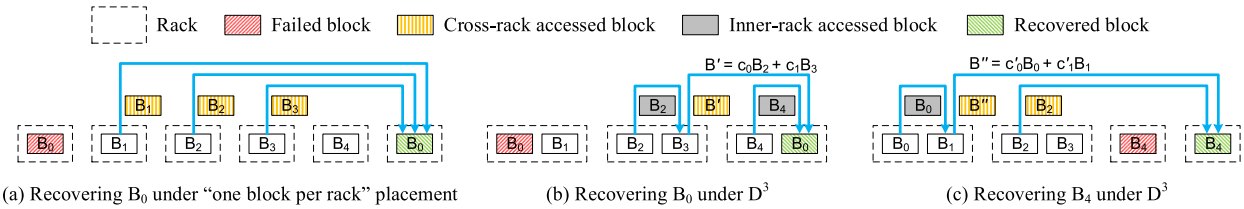


Fig. 2. Minimization of cross-rack blocks.

rack. The random distribution achieves uniform distribution of data blocks among all nodes with a huge number of stripes. However, DSSes rebuild lost blocks batch by batch for a long recovery queue due to limited available system resources, such as memory and CPU, etc. The random distribution leads to skew distribution of data blocks in a batch with limited stripes, and further induces random access and load imbalance for rebuilding blocks in a batch. Such random access and "local load imbalance" in a batch will slow down the rebuilding process.

This motivates us to design a new scheme for distributing blocks in DSSes based on erasure codes, which takes failure recovery speed into account. In particular, our design aims for the following three objectives.

- *Objective 1: Uniform Data and Parity Blocks Distribution.* Each node contains the same amount of data blocks and the same amount of parity blocks to achieve load balance. Many front-end applications generate intermediate temporary data and also profit from load balance data layout.
- *Objective 2: Minimal Cross-Rack Recovery Traffic.* Suppose that there are t nodes in the system. Let s_i $(0 \leq i \leq t-1)$ be the number of cross-rack accessed blocks for recovering the ith node. With a data layout against either a single rack failure or m node failures, we aim to minimize $S = \sum_{i=0}^{t-1} s_i$, the total number of cross-rack accessed blocks for recovering all the single-node failures, where $s_i = 0$ when the ith node fails, once for a node.
- *Objective 3: Locally Load-Balanced Recovery Traffic.* Under a single node failure, the reconstruction read/write loads are well balanced within a short range of successive stripes among the surviving racks, i.e., the racks not containing the failed node, and the read/write/computing loads are balanced among all nodes in the surviving racks, which will significantly speed up the recovery process and be also beneficial to front-end users' services.

3.2 D^3 Design – An Example

We now use a simple example of distributing the blocks of a $(3, 2)$-RS code to a storage system with $r = 5$ racks and $n = 3$ nodes in each rack to show the main idea of D^3. D^3 conducts block placement in the following three stages.

- *Minimizing Cross-Rack Accessed Blocks.* The lost blocks due to failures can only be recovered with other blocks in the same stripe. So the layout of blocks within a stripe determines the cross-rack traffic for recovery. We first design the layout of blocks within a stripe to achieve Objective 2.
- *Load Balance within a Rack.* Based on the layout of blocks within a stripe, we use an orthogonal array to place blocks of a cluster of stripes so that the blocks to be accessed for inner-rack block aggregation and the cross-rack accessed blocks for recovery are evenly distributed among nodes within a rack. We name a cluster of stripes as a *stripe region*.
- *Load Balance in All Racks.* We use another orthogonal array to distribute a group of stripe regions to racks to achieve Objective 1 and Objective 3.

3.2.1 Minimizing Cross-Rack Accessed Blocks

Let B_0, B_1, B_2, B_3, B_4 be the five blocks in a stripe of $(3, 2)$-RS code. Suppose we are to reconstruct the failed block B_0. With the "one block per rack" placement as shown in Fig. 2a, we need to access three blocks, say B_1, B_2, B_3. Furthermore, the recovered block of B_0 should be written into another rack different from those storing B_i $(0 \leq i \leq 4)$. So totally we access three blocks across racks.

With D^3, we first divide the five blocks into three groups as $\{B_0, B_1\}$, $\{B_2, B_3\}$, and $\{B_4\}$, and place each group in a rack with two blocks within a group in two different nodes for tolerating a single rack failure. We are still to recover B_0. Because RS codes satisfy the property of linearity, B_0 is a linear combination of B_2, B_3, B_4, say $B_0 = c_0B_2 + c_1B_3 + c_2B_4$, where the c_i's $(0 \leq i \leq 2)$ are the decoding coefficients specified by the given RS code. As shown in Fig. 2b, we can first read B_2 to the node containing B_3, then compute $B' = c_0B_2 + c_1B_3$, read B' and B_4 to a new node in the rack containing B_4, and at last recover B_0 as $B_0 = B' + c_2B_4$. So recovering B_0 only needs access one block across racks. Similarly, one block is to be accessed across racks to recover any of B_1, B_2, and B_3. However, recovering B_4 needs to access two blocks across racks, as shown in Fig. 2c. On average, the number of cross-rack accessed blocks for recovering a failed block under D^3 is $(1 \times 4 + 2 \times 1)/5 = 1.2$, which reaches the minimum with theoretical guarantee referred to Lemma 4. So the data layout within a stripe achieves Objective 2.

D^3 reaches the same reliability against node failures as the "one block per rack" placement, but it degrades the reliability against rack failures. Because the probability of double rack failures is negligible, e.g., HDFS also protects against single rack failure as default, D^3 is acceptable against single rack failure.

With grouping a stripe, we place each group into a rack and distribute the blocks within a group into different nodes in the corresponding rack. We use two orthogonal arrays, $\mathcal{A}$ and $\mathcal{A}'$, for rack-level and node-level distribution, respectively, to achieve load balance.

3.2.2 Load Balance Within a Rack

We first display node-level distribution for load balance among all nodes within a rack. Suppose there are three

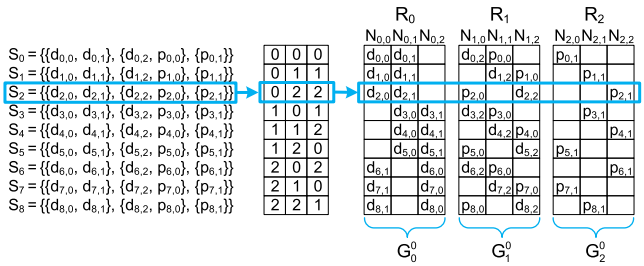


Fig. 3. Load balance within a rack.

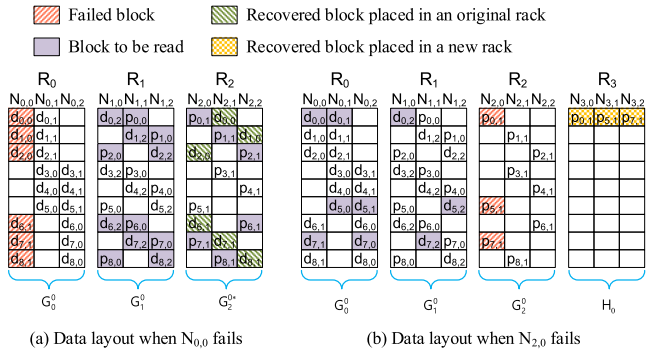


Fig. 4. Load balance within a rack when one node fails.

racks, R_0, R_1, R_2, for placing the three groups of (3, 2)-RS coded stripes, where R_i consists of three nodes, $N_{i,0}$, $N_{i,1}, N_{i,2}$, for $0 \leq i \leq 2$, as shown in Fig. 3c.

We select an OA$(3, 3)$, $\mathcal{A} = (a_{ij})_{9\times 3}$ (as shown in Fig. 3b), where $a_{ij} \in \{0, 1, 2\}$, as an auxiliary to distribute the blocks. Each column and each entry of $\mathcal{A}$ correspond to a rack and a node in the rack respectively. Because there are nine rows in $\mathcal{A}$, we place a cluster of nine stripes, $S_0, S_2, \ldots, S_8$, according to $\mathcal{A}$, as shown in Fig. 3a. We partition all blocks in S_i into three groups as above, $g_{i,0}, g_{i,1}, g_{i,2}$, for $0 \leq i \leq 8$, where there are two blocks in each of $g_{i,0}, g_{i,1}$, but only one block in $g_{i,2}$. According to $\mathcal{A}$, we place two blocks in $g_{i,j}$ to two nodes $N_{j,a_{ij}}$ and $N_{j,(a_{ij}+1) \bmod 3}$ in rack R_j for $j = 0, 1$ respectively, and place the block in $g_{i,2}$ to node $N_{2,a_{i2}}$ in rack R_2, as shown in Fig. 3c. For example, the third row of $\mathcal{A}$ is $(0, 2, 2)$. So we should place blocks $d_{2,0}$ and $d_{2,1}$ in the first group to $N_{0,0}$ and $N_{0,1}$ in R_0 respectively, $d_{2,2}$ and $p_{2,0}$ to $N_{1,2}$ and $N_{1,0}$ in R_1 respectively, and $p_{2,1}$ to $N_{2,2}$ in R_2. We find that the blocks in a cluster of nine stripes are evenly distributed among the nodes within a rack. We name a cluster of nine stripes as a *stripe region*, and we divide the ith stripe region into three *region-groups*, G_0^i, G_1^i, and G_2^i, where G_j^i is in rack R_j for $0 \leq j \leq 2$.

Note that there are two types of group-size in (3, 2)-RS coded stripes, say 2 of $g_{i,0}$ and $g_{i,1}$ and 1 of $g_{i,2}$. We can distinguish two types of region-groups with recovered blocks corresponding to whether the recovered blocks are placed in a new rack.

3.2.3 Load Balance in All Racks

Suppose node $N_{0,0}$ in rack R_0 fails (Fig. 4a). Since R_2 contains less blocks than R_0, we store the recovered blocks on the nodes in R_2, $N_{2,(a_{i2}+1) \bmod 3}$, for $i = 0, 1, 2, 6, 7, 8$. Thanks to Property 2 of orthogonal arrays, the blocks needed to be accessed in racks R_1 and R_2 are evenly distributed among the nodes in each rack, and the recovered blocks are also evenly distributed among the nodes in R_2. The changed region-group G_2^i is the first type of region-groups with recovered blocks, denoted as G_2^{i*}.

Suppose a node in R_2, say $N_{2,0}$, fails. Since the recovered blocks need to be placed in a rack different from R_2 and the first two racks R_0 and R_1 contain more blocks than R_2, we distribute the recovered blocks to three nodes in a new rack R_3 evenly in a round-robin way as shown in Fig. 4b, which forms another type of region-group with recovered blocks, denoted as H_i. Similarly, the blocks needed to be accessed in racks R_0 and R_1 are evenly distributed among the nodes in each rack, and the recovered blocks are also evenly distributed among the nodes in R_3.

We have distributed the blocks within a stripe region to reach load balance within a rack. Now we distribute $r(r-1) = 20$ stripe regions to a system of $r = 5$ racks to reach load balance among the nodes in all racks. We select an OA$(5, 4)$, $\mathcal{A}' = (a'_{ij})_{25\times 4}$ as shown in Fig. 5d to define the data placement of 20 stripe regions, where $a'_{ij} \in \{0, 1, 2, 3, 4\}$

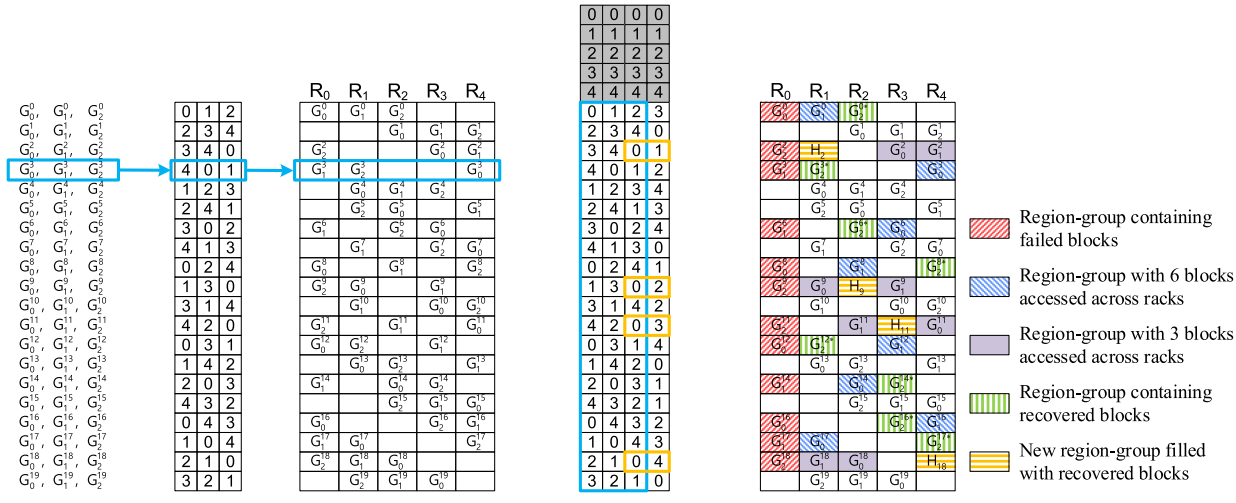


Fig. 5. Load balance in all racks.

corresponds to rack $R_{a'_{ij}}$. Note that the 5 columns in the first five rows of $\mathcal{A}'$ are identical. If we use the first five rows of $\mathcal{A}'$ to define the placement of a stripe region, then all groups in the stripe will be placed in the same rack, which does not tolerate any rack failure. So we only use the sixth to twenty-fifth rows to define the placement, denoted as another matrix $\mathcal{M} = (m_{ij})_{20\times 4}$, i.e., $m_{ij} = a'_{i+5,j}$.

We diagram the step of placing stripe regions to reach load balance among all nodes in all racks in Fig. 5, where each row of Fig. 5a corresponds to a stripe region, Fig. 5b shows the first three columns of $\mathcal{M}$ which define the placement of the region-groups, and Fig. 5c shows the placement. We distribute G_0^i, G_1^i, G_2^i to racks $R_{m_{i0}}$, $R_{m_{i1}}$, $R_{m_{i2}}$ respectively, for $0 \le i \le 19$. For example, $(m_{30}, m_{31}, m_{32}) = (4, 0, 1)$, we distribute G_0^3, G_1^3, G_2^3 to racks R_4, R_0, R_1 respectively, as shown in the fourth row of Fig. 5c. For any j $(0 \le j \le 2)$, the 20 region-groups G_j^i's $(0 \le i \le 19)$ are evenly distributed among all racks. Thus, data blocks and parity blocks are evenly distributed among all nodes, i.e., we achieve Objective 1.

Suppose that one node in rack R_0 fails, as shown in Fig. 5e. There are 12 region-groups in R_0 containing a failed block each. These region-groups in R_0 can be categorized into two types according to group size, G_j^i's for $j = 0, 1$ and G_2^i's. Because of the even distribution of region-groups, there are four $G_j^{i'}$s in R_0 for each of $j = 0, 1$ (see Fig. 5e), the recovery method for each is similar with Fig. 4a and corresponding region-group with recovered blocks G_2^i become G_2^{i*}. There are also four G_2^i's in R_0, and the each corresponding region-group with recovered blocks H_i (see Fig. 4b) is placed into rack $R_{m_{i3}}$.

From Figs. 4a, 2b and 4b, 2c, we find that a rack containing a region-group G_2^{i*} or H_i will induce six cross-rack blocks written into it. Furthermore, from Fig. 5e, we find that both of G_2^{i*}'s and H_i's are evenly distributed among the surviving racks. So the load of cross-rack write for recovering failed blocks is balanced among the surviving racks. Similarly, the load of cross-rack read for recovering failed blocks is also balanced among the surviving racks as shown in Fig. 5e. Since $\mathcal{A}'$ balances each type of region-groups (e.g., a type of region-groups as H_i's) involved in recovery among the surviving racks, the loads of read/write/computing for reconstructing the failed blocks are balanced among all nodes in the surviving racks. Thus, we achieve Objective 3.

4 The General Design of D^3

In this section, we present the general design of D^3. Suppose we are to deploy (k, m)-RS code to a system composed of r racks with n nodes each, i.e., $n \times r$ nodes in total. D^3 conducts block placement in three stages, data layout of a stripe, load balance within a rack, and load balance in all racks.

4.1 Data Layout of a Stripe

On one hand, in order to minimize cross-rack repair traffic, we place multiple blocks of a stripe in a rack at the expense of reducing rack-level fault tolerance. On the other hand, a rack may contain at most m blocks of the same stripe for tolerating a single rack failure. Thus, D^3 divides the $len = k + m$ blocks of a stripe into $N_g = \lceil len/m \rceil$ groups, $g_0, g_1, \ldots, g_{N_g-1}$, such that there are $Size_{max} = \lceil len/N_g \rceil$ blocks in each of the first $t = len \bmod N_g$ groups $g_0, g_1, \ldots, g_{t-1}$ and $Size_{min} = \lfloor len/N_g \rfloor$ blocks in each of the remaining $N_g - t$ groups $g_t, g_{t+1}, \ldots, g_{N_g-1}$. Specifically, let $B_0, B_1, \ldots, B_{k+m-1}$ be the $k + m$ blocks in a stripe. D^3 allocates $B_0, \ldots, B_{Size_{max}-1}$ to g_0, $B_{Size_{max}}, \ldots, B_{2\times Size_{max}-1}$ to $g_1, \ldots, B_{(t-1)\times Size_{max}}, \ldots, B_{t\times Size_{max}-1}$ to g_{t-1}, $B_{t\times Size_{max}}, \ldots, B_{t\times Size_{max}+Size_{min}-1}$ to g_t and so on. So the allocation of all blocks in a stripe into N_g groups is determined and unique.

Each group will be placed into a separate rack, as shown in Fig. 2b. We have the following two lemmas.

Lemma 1. *With the data layout of D^3, there are at most m blocks of the same stripe in a group.*

Lemma 2. *Suppose that $len = am + b$, where $a = \lfloor len/m \rfloor$ and $b = len \bmod m$. With the data layout of D^3, if $0 < b < m - 1$, then there are at least two groups, where there are no more than $m - 1$ blocks of the same stripe in each group.*

4.2 Load Balance Within a Rack

Now we are to place a group of stripes to reach load balance among the n nodes $(n \ge m)$ within a rack. Because there are n nodes in each rack, and we divide the blocks of each stripe into N_g groups, we need to distribute a cluster of n^2 stripes to the nodes in N_g racks to reach load balance among all nodes within a rack. Denote the n^2 stripes as $S_0, S_1, \ldots, S_{n^2-1}$, where S_i is divided into N_g groups $g_{i,0}, g_{i,1}, \ldots, g_{i,N_g-1}$ for $0 \le i \le n^2 - 1$. We select an OA(n, N_g), $\mathcal{A} = (a_{ij})_{n^2\times N_g}$, to define the block placement. The kth block in $g_{i,j}$ is placed to $N_{j,((a_{ij}+k) \bmod n)}$ in rack R_j for $0 \le k \le Size_{max}$ when there are $Size_{max}$ blocks in $g_{i,j}$; otherwise for $0 \le k \le Size_{min}$ when there are $Size_{min}$ blocks in it. Refer to Fig. 3c as an example. Based on the block placement, we have the following lemma.

Lemma 3. *Within a cluster of n^2 stripes, D^3 places the same number of data/parity blocks to each node in the same rack.*

4.3 Load Balance in All Racks

Denote a cluster of n^2 stripes as a *stripe region*. We now distribute $r(r-1)$ stripe regions to a system of r racks $(r > N_g)$ to reach load balance among all nodes in all racks. We select an OA$(r, N_g + 1)$, $\mathcal{A}'$, with its first r rows of all $N_g + 1$ columns being identical. We ignore the first r rows of $\mathcal{A}'$ and denote the submatrix of $\mathcal{A}'$ without the first r rows as $\mathcal{M}$, which is used to define block placement of $r(r-1)$ stripe regions. The last column of $\mathcal{M}$ is used for failure recovery.

Denote the ith stripe in the jth stripe region as S_i^j, where S_i^j is divided into N_g groups $g_{i,0}^j, g_{i,1}^j, \ldots, g_{i,N_g-1}^j$ for $0 \le i \le n^2 - 1, 0 \le j \le r(r-1) - 1$. According to the group division of stripes, we divide the jth stripe region into N_g *region-groups*, $G_0^j, G_1^j, \ldots, G_{N_g-1}^j$, where G_k^j consists of $g_{0,k}^j, g_{1,k}^j, \ldots, g_{n^2-1,k}^j$ for $0 \le k \le N_g - 1$. Suppose $\mathcal{M} = (m_{jk})_{(r(r-1))\times(N_g+1)}$. We place the kth region-group of the jth stripe region, G_k^j, into rack $R_{m_{jk}}$ for $0 \le j \le r(r-1) - 1, 0 \le k \le N_g - 1$. Refer to Fig. 5c as an example. The following theorem shows that D^3 achieves Objective 1.

Theorem 2. *Within $r(r-1)$ stripe regions, D^3 places the same number of data/parity blocks to each node in all racks.*

The following theorem shows the reliability of D^3.

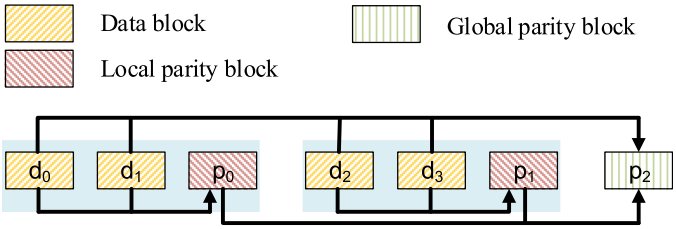


Fig. 6. A (4,2,1)-LRC stripe. ($k = 4$ data blocks, $l = 2$ local parity blocks and $r = 1$ global parity block.).

Theorem 3. *D^3 tolerates either m node failures or one rack failure.*

4.4 Extension to Locally Repairable Codes

A (k, l, g)-LRC [9] stripe contains k data blocks, l local groups and g global parity blocks. Fig. 6 gives an example of (4,2,1)-LRC. The storage overhead and repair traffic of LRCs are between traditional replication and RS codes. LRC keeps additional local parity blocks and introduces different reconstructing methods for different types of failed blocks. So, how to balance repair traffic becomes more difficult. In view of this, we extend D^3 to support LRC to improve repair performance. The block placement for LRCs contains similar stages with RS codes except skipping the first stage. The details are shown as follows.

4.4.1 Load Balance Across Nodes

In order to satisfy maximum rack-level fault tolerance, we place blocks in a region adhering to the following basic rules: (1) blocks within a local group placed in different racks, (2) all parity blocks placed in different racks. So, we select an $\mathrm{OA}(n, N_g^{lrc})$, $\mathcal{A} = (a_{ij})_{n^2 \times N_g^{lrc}}$, for distributing blocks in node level, where $N_g^{lrc} = Max\{\frac{k}{l} + 1, l + g\}$. The following strategy can achieve the above assignments: (1) assign a column for each parity block in a stripe; (2) for each data block in a local group, assign a column different from the one assigned to the local parity block. In this way, a small orthogonal array suffices by one column of $\mathcal{A}$ specifying the location for multiple blocks instead of a block within one stripe. Based on the Property 1 of OA, we achieve load balance for LRCs across nodes whether within a rack or not.

For example in Fig. 7, we assign the (4,2,1)-LRC stripes to a DSS with OA(3,3) in Fig. 7b. For illustrating corresponding rules, we use the last column of OA(3,3) as the example. We first look up the last column of OA(3,3), and then assign data blocks and global parity blocks, saying $d_{i,j}$ and $p_{i,2}$ for $0 \le i \le 8$ and $j = 1, 3$, to three racks, say R_1, R_3 and R_6, with each entry specifying the ids of nodes in each rack. Similarly for other block, we achieve uniform blocks distribution in Fig. 7c.

4.4.2 Load Balance Across Racks

Now, we will assign $N_g = k + l + g$ region-groups to r racks and keep rack-level load balance (without loss of generality, $r > N_g$). Similarly to Section 4.3, we will find an $\mathrm{OA}(r, N_g + 1)$, $\mathcal{A}'$, and denote the submatrix of $\mathcal{A}'$ without the first r rows as $\mathcal{M}$, the first N_g columns of $\mathcal{M}$ act as an addressing table for the $k + l + g$ region-groups of each (k, l, g)-LRC stripe region, and the last column provides the address for the reconstructed blocks against single node failure. Then we place $r(r-1)$ regions, each of which consists of n^2 stripes, to racks in cluster based on the selected $\mathcal{M}$ with $r(r-1)$ rows. The whole step is similar to the step of load balance in rack level for RS codes in Section 4.3. So based on the properties of OA, we achieve load balance for LRCs in rack level.

From the above designs, we get the following theorem:

Theorem 4. *D^3 places the same number of data blocks, local parity blocks and global parity blocks to each node in all racks for Locally Repairable Codes.*

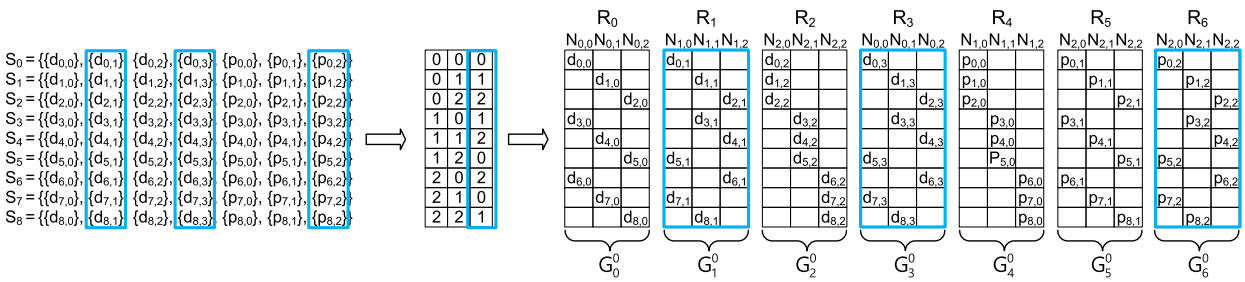
4.5 Valid Node Sizes and Rack Sizes of D^3

D^3 uses two orthogonal arrays to define its data layout. One is $\mathcal{A}$, for load balance within a rack, and the other is $\mathcal{A}'$, for load balance among nodes in all racks. From Theorem 1, given an integer n, let $n = p_1^{e_1} \dots p_v^{e_v}$ and $s = \min\{p_i^{e_i} + 1 : i = 1, \dots, v\}$, and there is an $\mathrm{OA}(n, s)$, where the $s - 1$ columns of the first r rows are identical [24]. From Definition 1, for any $s' \le s$, we can get an $\mathrm{OA}(n, s')$ just by taking any s' columns from $\mathrm{OA}(n, s)$.

The data layout of D^3 is defined by $\mathcal{A}$ and $\mathcal{A}'$, where n and r correspond to the number of nodes in a rack and the number of racks in a DSS respectively. From the construction of orthogonal arrays in [24], we know that there are plenty of $\mathrm{OA}(n, s)$'s for reasonable n, s. So we can obtain $\mathcal{A}$ and $\mathcal{A}'$ for practical configurations of DSSes.

5 Failure Recovery

In this section, we successively present recovery solutions for RS codes and LRCs based on D^3 against single node failure. Finally, we present a migration algorithm to maintain the original uniform data layout.



(a) Blocks in groups of stripes (b) An OA(3,3) (c) Data layout

Fig. 7. Load balance within a rack for (4,2,1)-LRC. $\{d_{i,0}, d_{i,1}, p_{i,0}\}$ and $\{d_{i,2}, d_{i,3}, p_{i,1}\}$ are in two local groups of LRC, $\{p_{i,2}\}$ is the global parity block. $\{p_{i,0}, d_{i,2}\}$, $\{d_{i,0}, p_{i,1}\}$ and $\{d_{i,1}, d_{i,3}, p_{i,2}\}$ use the three columns of OA to address ids of nodes, respectively.

5.1 Recovery for RS Codes

5.1.1 Recovery Process Within a Stripe

Let $B_0, B_1, \ldots, B_{len-1}$ be the $len = k + m$ blocks in a stripe. From Section 4.1, the len blocks are divided into $N_g = \lceil len/m \rceil$ groups such that there are $t = len \bmod N_g$ groups, each of which containing $Size_{max} = \lceil len/N_g \rceil$ blocks, and there are $N_g - t$ groups, each of which containing $Size_{min} = \lfloor len/N_g \rfloor$ blocks. Now assume one block of a stripe fails. Let μ be the average number of cross-rack accessed blocks for recovering the failed block. Suppose that $len = am + b$, where $a = \lfloor len/m \rfloor$ and $b = len \bmod m$. We have the following three cases.

1) If $b = 0$, we have $N_g = a$, $t = 0$, and $Size_{min} = Size_{max} = m$, i.e., each group contains m blocks. We use the $k = (a-1)m$ blocks from the $a-1$ surviving groups to reconstruct the failed block as follows. In each surviving group, the node containing the block with the largest subscript reads the other $m-1$ blocks in the same group and performs the inner-rack block aggregation, and then sends the aggregated block to some node in a new rack, where the failed block is reconstructed with the $a-1$ aggregated blocks. So we have $\mu = a - 1$ in this case.
2) If $0 < b < m - 1$, we have $N_g = a + 1$. By Lemma 2, there are at least two groups which contains no more than $m-1$ blocks. Suppose that R_x is the rack with largest subscript x which holds a surviving group with $z \leq m - 1$ blocks. We select $k - z$ blocks with the smallest subscripts from the $a-1$ surviving groups excluding the group in R_x. In each of the surviving groups containing any of these $k - z$ selected blocks, the node containing the selected block with the largest subscript in this group reads the other selected blocks in the same group and performs the inner-rack block aggregation, then sends the aggregated block to a node N_y in rack R_x, where N_y does not contain any of the z blocks in R_x. N_y uses the $a-1$ aggregated blocks and the z blocks in R_x to compute the failed block. So we have $\mu = a - 1$ in this case.
3) Otherwise $b = m - 1$, then $N_g = a + 1$, $t = a$, $Size_{max} = m$, $Size_{min} = m - 1$. There are a groups, each of them containing m blocks, and one group containing $m-1$ blocks.

   (3.1) If the failed block is in a group containing m blocks, then there are $a-1$ surviving groups, each containing m blocks. In each surviving group containing m blocks, the node containing the block with the largest subscript reads the other $m-1$ blocks in the same group and performs the inner-rack block aggregation, and then sends the aggregated block to node N_x, where N_x is in the rack storing the group with $m-1$ blocks and N_x does not contain any of the $m-1$ blocks in this rack. N_x uses the $a-1$ aggregated blocks and the $m-1$ blocks in the local rack to recover the failed block. It reads $a-1$ blocks across racks. Refer to Fig. 2b as an example.

   (3.2) Otherwise, the failed block is in the group containing $m-1$ blocks. We select $k = am - 1$ blocks with the k smallest subscripts from the a surviving groups, i.e., only excluding the block with the largest subscript in the a surviving groups. In each surviving group, the node containing the selected block with the largest subscript in this group reads the other selected blocks in the same group and performs the inner-rack block aggregation, and then sends the aggregated block to some node N_x in a new rack. N_x uses the a aggregated blocks to recover the failed block. It reads a blocks across racks. Refer to Fig. 2c as an example.

   So in case of $b = m - 1$, we have $\mu = [(a-1)(k+1) + a(m-1)]/(k+m)$.

Furthermore, we have the following lemma.

Lemma 4. *Given a (k, m)-RS code, suppose that $len = k + m = am + b$, where $a = \lfloor len/m \rfloor$ and $b = len \bmod m$. Then, with the data layout of D^3, the average number of cross-rack accessed blocks for recovering a failed block within a stripe is*

$$\mu = \begin{cases} \dfrac{(a-1)(k+1) + a(m-1)}{k+m}, & \text{if } b = m-1; \\ a - 1, & \text{otherwise,} \end{cases} \tag{1}$$

and reaches the minimum average number of cross-rack accessed blocks for recovering a failed block for data layouts against a single rack failure.

5.1.2 Recovery Process Within a Stripe Region

Based on the recovery process within a stripe, we place the recovered blocks to nodes as follows.

1) Suppose we are to place a recovered block B of stripe S_i to a node in its original rack R_x. Suppose that the block with the largest subscript stored in R_x and belonging to S_i is placed to node $N_{x,j}$. Then D^3 places B to node $N_{x,((j+1) \bmod n)}$ in R_x. Refer to Fig. 4a as an example.
2) Otherwise, the recovered blocks are placed into a new rack. D^3 places the recovered blocks to nodes of the new rack in the round-robin order. Refer to Fig. 4b as an example.

Based on the recovery process within a stripe region, we have the following lemma.

Lemma 5. *To recover a failed node, D^3 achieves load balance among all nodes within a rack in terms of overheads of read, write, and computing respectively.*

5.1.3 Recovery Process in $r(r-1)$ Stripe Regions

The repair loads are balanced among all nodes in surviving racks in $r(r-1)$ stripe regions with D^3. From Section 4.2, we know that each stripe region is divided into N_g region-groups, $G_0, G_1, \ldots, G_{N_g-1}$.

1) If the recovered blocks of G_i^j in the jth stripe region are placed in the rack with G_x^j, then G_x^j changes and the changed one is denoted as G_x^{j*}, referring to Fig. 5e as an example.
2) Otherwise, the recovered blocks of G_i^j are placed in a new rack, forming a new region-group, say H_j. Then H_j is placed into rack $R_{m_{jN_g}}$, where m_{jN_g} is the (j, N_g)-entry of $\mathcal{M}$, referring to Fig. 5e as an example.

The following two theorems show that D^3 achieves Objective 2 and Objective 3 respectively.

Theorem 5. *With the data layout of D^3, the number of cross-rack accessed blocks for recovering a single node failure is minimized for data layouts against a single node failure.*

Theorem 6. *D^3 achieves load balance among surviving racks in terms of overheads of cross-rack read and write. It also achieves load balance among all nodes in surviving racks in terms of overheads of read, write, and computing respectively, for single-node failure recovery.*

5.2 Recovery for Locally Repairable Codes

From Section 4.4, we used the first N_g columns of $\mathcal{M}$ to guarantee load balance in rack level. For single node failure recovery, we will choose a new rack to place the reconstructed block for per failed block. The ids of new racks are chosen based on the last column of $\mathcal{M}$. There are two types of failed blocks:

- *Data blocks/local parity blocks.* For a failed data block or local parity block, it can be reconstructed by other blocks within the local group and read $\frac{k}{l}$ blocks across racks. To place the reconstructed blocks, we first choose racks based on the last column of the $\mathcal{M}$, and then choose nodes within the racks in a round-robin way. The chosen nodes execute reconstructed tasks and store the obtained reconstructed blocks.
- *Global parity blocks.* When reconstructing a failed global parity block, it needs read all $l + g - 1$ parity blocks for reconstruction. The way choosing the targeted racks and nodes for the reconstructed blocks is the same as the case above.

On one hand, in every region, $\mathcal{A}$ guarantees balanced recovery traffic in node level. On the other hand, for load balance of recovery traffic among multiple regions, the last column of $\mathcal{M}$ gives uniform choices from surviving racks, and the first N_g columns of $\mathcal{M}$ guarantee recovery traffic load balance in rack level. Therefore, we get the following theorem:

Theorem 7. *When recovering single-node failure, D^3 achieves load balance among surviving nodes in terms of read, write, and computing for Locally Repairable Codes respectively .*

5.3 Maintain the Original D^3 Data Layout

D^3 achieves minimum cross-rack traffic and optimal load balance in node and rack level when recovering a single node failure. So we need do some maintenance for keeping the data layout to guarantee excellent recovery performance. When one recovery process completes, the data layout may not keep the group rules, i.e., $t = len \bmod N_g$ groups with $Size_{max}$ blocks and the remaining with $Size_{min}$ blocks (for LRC, $Size_{min} = Size_{max} = 1$), and some rack may hold less blocks than the other racks because the rack with failed node is not used to place recovered blocks. Based on the above two reasons, we get an interim data layout after completing one recovery. Before failures occur again, we need migrate part of data to the relived node (new online node or new replaced node in the rack with failed node), of course, which can be delayed to leisure state of DSS. In order to minimize the influence on front-end applications in DSSes, we design the migration strategy as following:

- *Batch-by-batch migration.* We need migrate all of the recovered blocks to the relived node to guarantee high recovery performance next time. Because storage capacity of a node can get dozens of TB [13], we need migrate massive data, which may cause network congestion, packet loss and so on, of course, and the front-end requests also gets performance degradation. To minimizing interference on other services, we split the migration operation to multiple batches and migrate the recovered blocks belonging to the same type stripe region.
- *Minimum data traffic of one batch.* For each batch of migration, we need to achieve optimal tradeoff between migration speed and data traffic. Fast migration means massive data traffic in one batch, but heavy cross-rack migration traffic causes scarce source occupation, such as bandwidth, and negative impacts on front-end applications. In each batch, we choose all the recovered blocks in $n - 1$ region-groups with recovered blocks of the same type from $n - 1$ racks without failed node, which can achieve balanced migration traffic with lowest migration load.

For the above migration strategy, we have the following theorem to guarantee the desired properties.

Theorem 8. *The migration gets optimal tradeoff between migration speed and data traffic, and data traffic in the process is load balance among $r - 1$ racks in DSSes.*

For an example in Fig. 5e, we execute migration in three batches: $[H_2, H_9, H_{11}, H_{18}]$, $[G_2^{3*}, G_2^{0*}, G_2^{14*}, G_2^{8*}]$ and $[G_2^{12*}, G_2^{6*}, G_2^{16*}, G_2^{17*}]$, and each batch causes 4×3 or 4×6 blocks cross-rack transferring. In summary, we achieve load balance in migration to maintain the original D^3.

6 Performance Evaluation

6.1 Evaluation Methodology

We implement D^3 in *HDFS Erasure Coding* (HDFS-EC) [16] shipped with HDFS in Apache Hadoop 3.1.x. HDFS-EC is a module integrated in HDFS to support erasure-coded solutions with built-in erasure coding policies including the (2,1), (3,2), and (6,3)-RS codes. An HDFS cluster consists of a *NameNode* and a number of *DataNodes*. The NameNode stores the metadata (e.g., block locations) for managing file operations, while the DataNodes store data. In terms of overhead, D^3 uses orthogonal arrays to define block placement and the deterministic distribution of data blocks. So it only needs to store the orthogonal arrays in the metadata server, and its memory cost only depends on the number of racks and nodes, which is around several KB for dozens of nodes. Besides, other overheads like CPU cost for addressing is negligible compared to the network transfer cost, because network bandwidth is usually considered as a scarcer resource in DSSes.

We conduct our evaluation on a cluster of 28 machines, each of which is configured with a quad-core 3.40 GHz Intel Core i5-7500, 8 GB memory, and a Seagate ST1000DM010-2EP102 7200 RPM 1 TB SATA hard disk, and runs Ubuntu 16.04. One machine runs as the NameNode and each of the

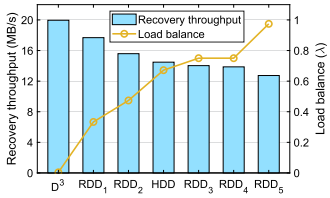


Fig. 8. Recovery under RS.

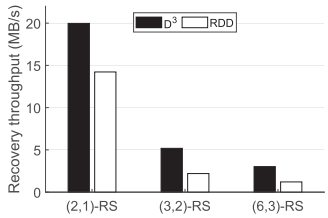


Fig. 9. RS codes configuration.

other machines runs as a DataNode. All DataNodes are divided into some racks, where all machines within a rack are connected via a UTT SG108V 8-Port 1000 Mb/s Ethernet switch and all racks are connected via a D-LINK DES-1024D 24-Port 100 Mb/s Ethernet switch to simulate practical systems. The available cross-rack bandwidth per node is only 1/20 to 1/5 of the inner-rack bandwidth in practical systems [18], [19], and more specifically, the available inner-rack bandwidth per node is 1,000 Mb/s as described in [19]. Therefore, the settings of the inner-rack and cross-rack bandwidths in our experiments are reasonable.

We take the commonly used *random data distribution* (RDD) in practical systems as baseline. In our experiments, we make RDD randomly distribute blocks of each stripe among all nodes, while ensuring single-rack fault tolerance. The recovery solution of RDD is that it randomly chooses k surviving blocks of a stripe and sends them to a randomly selected node excluding the nodes containing the blocks of the same stripe for recovery. We write 1,000 stripes of blocks with D^3 and RDD respectively, and conduct our evaluation with different settings. To evaluate the recovery performance, we randomly pick one DataNode and erase all of its blocks, and then measure the *recovery throughput*, defined as the total volume of failed blocks being repaired over the total recovery time. The presented results are averaged over five runs.

6.2 Experimental Results

By default, the DSS consists of eight racks with three DataNodes each, and the block size is set as 16 MB using (2,1)-RS code. We also conduct experiments with different erasure codes and rack configurations.

6.2.1 Recovery Performance

Experiment 1 (Repair Load Balance). Because the cross-rack data transfer is crucial to the system performance, we first evaluate the distributions of cross-rack repair traffic loads of RDD and D^3. In our experiments, each port of the switch connecting the racks is full-duplex, with 100 Mb/s upstream and 100 Mb/s downstream available simultaneously. For the port connecting to the ith surviving rack ($0 \leq i \leq 6$), let L_i, L_i' be the upstream load from and the downstream load to this port respectively. Let $L_{max} = \max_{0 \leq i \leq 6}\{L_i, L_i'\}$ and $L_{avg} = \sum_{0 \leq i \leq 6}(L_i + L_i')/14$. Then we measure the *load imbalance* by $\lambda = (L_{max} - L_{avg})/L_{avg}$.

We run RDD with five groups, and each group has a different random data distribution with a randomly chosen failed node. In each group, we run the experiments five times with the same data distribution and the same failed node, and average the recovery throughput and the corresponding load imbalance metric λ, as shown in Fig. 8, where RDD_i is the ith group and the results are sorted by λ. From Fig. 8, we find the repair load with D^3 is balanced due to our deterministic data layout, but it is constantly imbalance with RDD. In the five groups of experiments, λ varies from 0.33 to 0.97, i.e., L_{max} is at least 1.33 times of L_{avg}, which shows severe imbalance of repair load. RDD achieves uniform distribution of blocks as long as the number of stripes is large enough. However, due to the limited resources in a node, such as memory and CPU, etc, the reconstruction is executed batch by batch. As a result, the distribution of blocks with RDD is severely skewed in a batch which has limited number of stripes. We can also find from Fig. 8 that the repair load imbalance significantly slows down the recovery process and the recovery throughput decreases when λ increases. D^3 increases the recovery throughput by 35.92 percent on average, compared with RDD.

We also evaluated the *hash-based data distribution* (HDD) for comparison. HDD uses a pseudo-random hash function to map an input value (typically a block or data object identifier) to a list of storage devices, and it generates a pseudo-random (but deterministic) data distribution [15], [27], [28]. In the implementation of HDD, we use the Jenkins hash function [29], which is adopted by data placement algorithms such as CRUSH [15], to map blocks to nodes, while ensuring single-rack fault tolerance. HDD will reselect nodes using a modified input (by following reselection behavior of CRUSH [15]) for three different reasons: (1) if a block is mapped to a node already used in the current stripe, (2) if a block is mapped to a node which cannot ensure single-rack fault tolerance, or (3) if a node is failed. The recovery throughput and the corresponding load imbalance metric λ of HDD are also shown in Fig. 8, where marked as "HDD". D^3 increases the recovery throughput by 37.83 percent, compared with HDD.

Experiment 2 (Erasure-coded Configuration). We use three HDFS build-in (2,1), (3,2), and (6,3)-RS codes to evaluate the recovery throughput of D^3 with different erasure code settings, shown in Fig. 9. We find that the recovery throughput decreases as the stripe size increases from (2,1), (3,2) to (6,3)-RS. This is because more blocks participate in recovering one failed block with larger stripe size, which induces heavier load and thus reduces the recovery throughput. We also find that D^3 outperforms RDD more significantly as the stripe size increases from (2,1), (3,2) to (6,3)-RS code. The recovery throughput of D^3 with (2,1), (3,2), and (6,3)-RS codes are 1.40, 2.36, and 2.49 times of which of RDD respectively. This is because with more blocks in a rack as (3,2) and (6,3)-RS codes, D^3 aggregates more blocks into an aggregated one and saves more cross-rack traffic.

Experiment 3 (Degraded Read). A client (i.e., a HDFS node), which tries to read a lost data block that has not been repaired, will trigger a *degraded read* to repair the unavailable

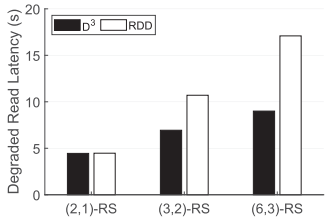


Fig. 10. Degraded read latency.

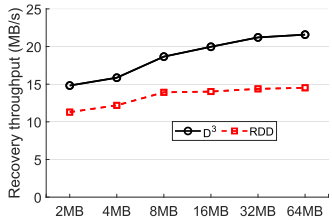


Fig. 12. Block size.

block. To evaluate the performance of degraded read, we randomly erase one data block and randomly choose a client to read the block, then measure the *degraded read latency*, defined as the time from issuing a read request until the failed block is repaired at the client. Fig. 10 shows the degraded read latency for D^3 and RDD with (2,1), (3,2), and (6,3)-RS codes. We find that the degraded read latencies of D^3 and RDD are almost identical for (2,1)-RS code. This is because each block of a single (2,1)-RS stripe is placed in a distinct rack to achieve rack-level fault tolerance for both D^3 and RDD. With (3,2) and (6,3)-RS codes, the degraded read latencies of D^3 are reduced by 35.16 and 47.34 percent respectively, compared with RDD. This is because D^3 aggregates blocks within racks and minimizes cross-rack accessed blocks within a single stripe. So D^3 induces less cross-rack traffic for recovery and has a shorter degraded read latency.

We also plot the *data recovery rate*, defined as the size of the failed block divided by the degraded read time, as shown in Fig. 11. We find that, for each of (2,1), (3,2), and (6,3)-RS codes, the data recovery rate in Fig. 11 is smaller than the recovery throughput in Fig. 9, especially much smaller for (2,1)-RS code. This is because degraded read only repairs a single block, while node recovery needs to repair many blocks, which induces more racks and nodes participating in the recovery with a larger degree of parallelism.

6.2.2 Sensitivity to Internal Parameters

Experiment 4 (Block Size). We now study the performance of D^3 with different block sizes. We fix the distribution of RDD with $\lambda = 0.75$ and change the block size. Fig. 12 shows the recovery throughput for the block sizes from 2 to 64 MB. We find that with both D^3 and RDD, the recovery throughput almost increases along with the increase of block size, which reaches the maximum when the block size is 32 MB. But compared to RDD, D^3 increases the recovery throughput almost with a consistent ratio, about 39.57 percent, with different block sizes.

Experiment 5 (Cross-Rack Bandwidth.) We now present the performance of D^3 with different cross-rack bandwidths. We fix the distribution of RDD with $\lambda = 0.33$ and 0.75, and change the cross-rack bandwidth by connecting the racks via a D-LINK DES-1024D 24-Port 100 Mb/s Ethernet switch or a TP-LINK TL-SG1016DT 16-Port 1,000 Mb/s Ethernet switch. Fig. 13 shows the recovery throughput. On average, D^3 increases 27.82 and 18.10 percent of the recovery throughput compared to RDD with cross-rack bandwidth of 100 and 1,000 Mb/s, respectively. Furthermore, we find that the recovery throughput increases significantly when the cross-rack bandwidth increases from 100 to 1,000 Mb/s, which implies that the cross-rack traffic is crucial to the recovery performance.

Experiment 6 (Number of Racks). We fix the number of nodes in each rack as three to evaluate the performance of D^3 with five, seven, and nine racks, as shown in Fig. 14. We find that the recovery throughput increases with more racks. This is because the cross-rack bandwidth is crucial to recovery throughput, and the total cross-rack bandwidth increases with more racks, which benefits the failure recovery and increases the recovery throughput. We also find that D^3 increases the recovery throughput more evidently with more racks, 1.21, 1.49, and 1.64 times of which of RDD with five, seven, and nine racks respectively. This is because with more racks, RDD suffers more severe imbalance of repair load.

Experiment 7 (Number of Nodes per Rack). We fix the number of racks as five, and set three, four, and five nodes per rack to evaluate the performance of D^3, as shown in Fig. 15. When the number of nodes per rack varies from three to five, the recovery throughput with D^3 and RDD does not significantly vary. The reason is that the cross-rack bandwidth, which is crucial to the recovery throughput, will not change when the number of nodes per rack varies.

6.2.3 Recovery Performance of LRCs

In this experiment, we study the performance of D^3 for Locally Repairable Code (LRC) [9]. The hadoop 3.1.x currently does not support LRC policies, but we implement and mimic a representative LRC policy in HDFS, which is (4,2,1)-LRC in the previous example, and we use it to compare the recovery throughput of D^3 and RDD. D^3 uses OA(3,3) and OA$(8, 4)$ to address node-level locations and place regions in rack level, respectively.

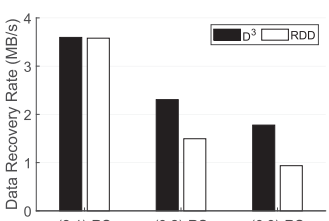


Fig. 11. Data recovery rate.

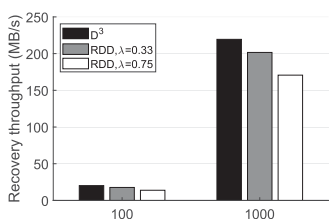


Fig. 13. Cross-rack bandwidth.

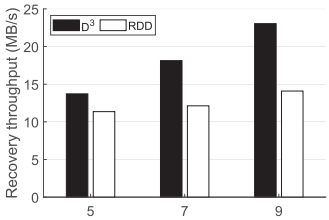


Fig. 14. Number of racks.

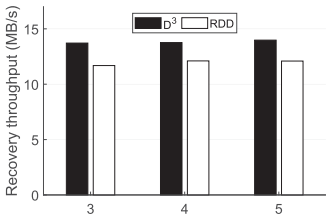


Fig. 15. Number of nodes per rack.

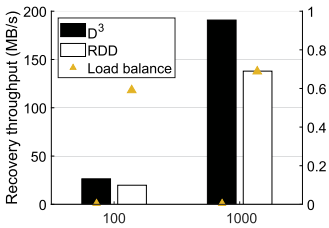


Fig. 16. Recovery under LRC.

*Experiment 8 (Recovery performance under LRC).*Fig. 16 shows the recovery throughput of D^3 and RDD over (4,2,1)-LRC with different cross-rack bandwidth configuration. When the cross-rack bandwidth is 100 and 1,000 Mb/s and the load balance λs of RDD are 0.5909 and 0.6879 respectively. D^3 increases 40.23 and 38.35 percent of the recovery throughput compared with RDD, respectively. The most primary reason is that D^3 achieves uniform distribution of recovery traffic, while random network traffic distribution of RDD in recovery causes the network-bandwidth proportions of rack-level switch ports are unbalanced. From another point of view, D^3 also balances computing tasks to different nodes, so it can also uniformly allocate CPU source and memory usage among surviving nodes.

Experiment 9 (Impact of block size under LRC). Block sizes are different for diverse users' demand and document types. In Fig. 17, we first take fixed $\lambda = 0.5909$ for RDD, and then we study the impact of block size in recovery under D^3 and RDD. With the increase of block size, the recovery throughput of D^3 and RDD both increases. Compared with RDD, D^3 increases recovery throughput over LRC by $20.13\% \sim 61.10\%$, and 31.98 percent on average. The reason is that reconstructed tasks for D^3 are uniformly distributed in node level, causing cross-rack recovery traffic uniformly distributed among racks.

6.2.4 Benchmark Tests Performance

A DSS also usually needs support many front-end users' responses, whose performance needs to be prioritized. In the experiment, we evaluate four hadoop benchmarks, namely *Pi*, *Terasort*, *Wordcount* and *Grep*, which will generate and store temporary data in HDFS depending on data layout in the tasks execution phase. The failure recovery of DSSes mainly competes cross-rack network bandwidth and CPU source with the four benchmarks. We still take the representative (2,1)-RS code as storage policy. The following describes characteristics of the four workloads, and the detailed configuration is shown in Table 2.

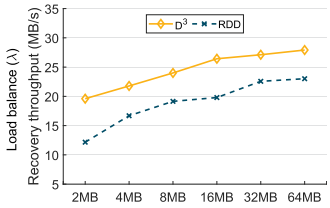


Fig. 17. Block size under LRC.

TABLE 2
Hadoop Benchmarks

Workloads	Application domain	Data source	Configuration
Pi	CPU intensive	Autogeneration	100 maps with 100m samples
Terasort	CPU/Network intensive	Table	Generating 5m records by TeraGen
Wordcount	Network intensive	Text	Randomly generating 100m words
Grep	Network intensive	Text	Randomly generating 100m words

- *Pi*. The map/reduce program uses a BBP-type method [30] to compute exact binary digits of π, the approximation algorithm needs lots of computing source, and the more tasks execution times, the more precise numerical value of π [31].
- *Terasort*. It is a standard map/reduce sort, except for a custom partitioner that uses a sorted list of N-1 sampled keys to define the key range for each reduce. It first samples the input and computes the input distribution, and then writes the list of keys into HDFS. TeraGen command generates the large random input data for Terasort [32].
- *Wordcount*. It reads the text input files, breaks each line into words and counts them. The output is a locally sorted list of words and the counts of how often they occurred [32], [33].
- *Grep*. It is an example of Hadoop Map/Reduce application. It extracts matched strings provided by users from text files and sorts matched strings by their frequency [32].

*Experiment 10 (Benchmark tests in normal).*From Fig. 18, we find D^3 shows better support to front-end users' responses when running the four workloads. The reason is that D^3 achieves a uniform data distribution for the intermediate data when running MapReduce tasks, which benefits distribution of network traffic when accessing temporarily stored data across nodes in HDFS. More specifically, for Pi, it mainly occupies CPU source of DataNodes and involves little network traffic, so D^3 doesn't show significant advantage when compared with RDD. Terasort and Wordcount need more cross-node network traffic, so more advantage is

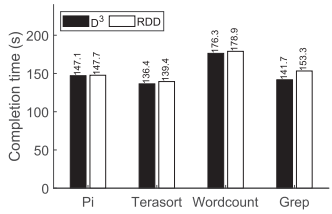


Fig. 18. Normal state.

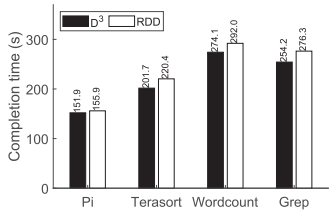


Fig. 19. Recovery state.

shown better because D^3 stores more uniform data. For Grep, which frequently needs extract and sort matched strings, it causes heavy cross-node traffic, so the system performance based on D^3 is improved up to 7.57 percent.

*Experiment 11 (Benchmark tests in recovery).*We first write 3000 stripes and then randomly erase one node to mimic single node failure. In the next moment, we run the Map-Reduce tasks with the same configuration in Table 2 from Namenode, and task execution process of the four benchmarks is in recovery state of DSSes. In Fig. 19, the recovery of D^3 only causes 3.26 percent performance reduction when compared with normal-state performance for Pi. The reason is that D^3 uniformly distributes reconstruction tasks to computing nodes with uniform CPU and memory occupation. For network-intensive Terasort, Wordcount and Grep, because of balanced network bandwidth among racks and nodes, D^3 reduces completion time by 8.48, 6.13 and 8.00 percent respectively when compared with RDD.

7 Related Work

*Data Layout Optimization.*Data layout refers to how to place data/parity blocks or replicas across disks or servers, which plays a critical role in both performance and reliability of RAIDs and DSSes. There are some approaches to improve RAID performance by means of data layouts [17], [34], [35], [36]. Parity declustering was first proposed by Muntz and Lui [34] as a data layout technique to speed up the failure recovery process and provide highly-available arrays. It requires less additional load on surviving disks for data reconstruction and the rebuild read load is balanced among all surviving disks. Parity declustering was further realized by Holland and Gibson [35] based on balanced incomplete block designs in practical systems. Although parity declustering spreads rebuild reads to surviving disks, rebuild is still capped by the write speed of the replacement disk. RAID+ [17] spreads both rebuild read and write to the surviving disks, and as a result, it speeds up data recovery for RAIDs. However, the data layouts of RAID+ for RAIDs are not efficient for failure recovery in DSSes. Unlike RAIDs, where the organization of disks is flat, DSSes typically connect storage nodes with a tree-structured topology. Therefore DSSes often exhibits the property of heterogeneous bandwidth, where the available inner-rack bandwidth is considered to be sufficient, while cross-rack bandwidth is often over-subscribed. Furthermore, in DSSes, each server has both storage resources and computing power, so every server can participate in encoding/decoding, while for RAIDs, computing can only be performed on the server that connecting the disks.

For data layouts in DSSes, EAR [21] is a placement algorithm for efficient transfer from replication to erasure coding. There are also some pseudo-random hash-based data placement approaches for scalable storage systems. Brinkmann *et al.* [28] use pseudo-random hash-functions for evenly distributing and efficiently locating data in dynamically changing SANs. Chord [37] uses a variant of consistent hashing [38] to assign keys to storage nodes for efficient adaption to dynamic change of nodes in systems. SCADDAR [39] uses a pseudo-random placement to distribute continuous media blocks across all disks, and it minimizes block movement in case of disk scaling. RUSH [27] and CRUSH [15] utilize pseudo-random hash-based functions to map replicated objects to a scalable collection of storage devices. They are probabilistically optimal in distributing data evenly and minimizing data migration when new storage devices are added to systems. These hash-based data placement strategies are designed for improving scalability in dynamic distributed environments. They all use pseudo-random hash functions to generate pseudo-random data distribution, while D^3 presents a deterministic data distribution to speed up data reconstruction.

Erasure Codes for Fast Recovery. RS codes [8] are the most popular erasure codes that are widely deployed in real storage systems [1], [2], [7]. Rotated RS codes [40] convert the one-row RS codes into multi-row codes and optimize the coding chains, which induces fewer data reads for faster degraded read. Regenerating codes [41] minimize the repair traffic by allowing other surviving nodes to send computed data for data reconstruction, and achieve an optimal tradeoff between storage redundancy and repair traffic. In particular, minimum-storage regenerating (MSR) codes [41] are MDS, and they minimize the repair traffic subject to the minimum storage redundancy. Rashmi *et al.* [42] propose a new MSR code construction that also minimizes the recovery I/Os. A class of erasure codes called LRCs [5], [9] are proposed to reduce the repair traffic, with at least 25 to 50 percent additional parities, and hence they are non-MDS. HACFS [43] dynamically switches between two different erasure codes according to workload changes so as to balance storage overhead and recovery performance. Our D^3 is a data placement scheme which deterministically distributes blocks into DSSes based on erasure codes to improve repair performance.

8 Conclusion

This paper proposes D^3, a *deterministic data distribution* scheme which significantly speeds up failure recovery. It defines the data distribution with orthogonal arrays, which reaches even distribution of data/parity blocks among nodes, balanced repair traffic among nodes and racks, and the minimal cross-rack repair traffic against a single node failure. It provides good support to MDS codes and LRCs.

Experimental results show that D^3 significantly speeds up the failure recovery process and efficiently supports front-end users' applications, when compared with random data distribution.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1003204, and in part by National Nature Science Foundation of China under Grant 61832011 and Grant 61772486. A preliminary version [44] of this article was presented at 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS'19). In this journal version, we extend normal data placement and recovery algorithmic of D^3 to Locally Repairable Codes [5], [9], [10], provide efficient strategy to maintain the original D^3 data layout after recovery, and conduct more experimental evaluation.

References

[1] D. Ford *et al.*, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 61–74.
[2] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. 5th USENIX Workshop Hot Topics Storage File Syst.*, 2013, Art. no. 8.
[3] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.
[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[5] C. Huang *et al.*, "Erasure coding in windows azure storage," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 15–26.
[6] S. Muralidhar *et al.*, "f4: Facebook's warm BLOB storage system," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 383–398.
[7] Facebook, "HDFS-RAID," 2011. [Online]. Available: https://wiki.apache.org/hadoop/HDFS-RAID
[8] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[9] M. Sathiamoorthy *et al.*, "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.
[10] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg, "On fault tolerance, locality, and optimality in locally repairable codes," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 865–877.
[11] Breaking resiliency trade-offs with locally decodable erasure codes, 2019. [Online]. Available: https://www.vastdata.com/, VASTDATA Inc
[12] B. Calder *et al.*, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 143–157.
[13] Z. Wang, G. Zhang, Y. Wang, Q. Yang, and J. Zhu, "Dayu: Fast and low-interference data recovery in very-large storage systems," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 993–1008.
[14] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.
[15] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. ACM/IEEE Conf. Supercomput.*, 2006, pp. 31–31.
[16] Hadoop, "HDFs erasure coding," 2018. [Online]. Available: https://hadoop.apache.org/docs/r3.1.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html
[17] G. Zhang, Z. Huang, X. Ma, S. Yang, Z. Wang, and W. Zheng, "RAID+: Deterministic and balanced data distribution for large disk enclosures," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 279–294.
[18] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.
[19] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, 2009.
[20] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 231–242, 2013.
[21] R. Li, Y. Hu, and P. P. Lee, "Enabling efficient and reliable transition from replication to erasure coding for clustered file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2500–2513, Sep. 2017.
[22] S. Xu *et al.*, "Single disk failure recovery for X-code-based parallel storage systems," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 995–1007, Apr. 2014.
[23] L. Xu, M. Lyu, Z. Li, Y. Li, and Y. Xu, "Deterministic data distribution for efficient recovery in erasure-coded storage systems," 2020, *arXiv: 2004.03998*.
[24] A. S. Hedayat, N. J. A. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*. Berlin, Germany: Springer, 2012.
[25] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, vol. 16. Amsterdam, The Netherlands: Elsevier, 1977.
[26] D. Stinson, *Combinatorial Designs: Constructions and Analysis*. Berlin, Germany: Springer, 2007.
[27] R. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, Art. no. 96.
[28] A. Brinkmann, K. Salzwedel, and C. Scheideler, "Efficient, distributed data placement strategies for storage area networks," in *Proc. 12th Annu. ACM Symp. Parallel Algorithms Archit.*, 2000, pp. 119–128.
[29] Hash functions for hash table lookup, 1997. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html
[30] Wikipedia, "Bailey–Borwein–Plouffe formula," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Bailey%E2%80%93Borwein%E2%80%93Plouffe formula
[31] S. Wei, Y. Li, Y. Xu, and S. Wu, "DSC: Dynamic stripe construction for asynchronous encoding in clustered file system," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[32] M. Malik *et al.*, "Big vs little core for energy-efficient hadoop computing," *J. Parallel Distrib. Comput.*, vol. 129, pp. 110–124, 2019.
[33] M. Malik, S. Rafatirad, and H. Homayoun, "System and architecture level characterization of big data applications on big and little core server architectures," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 3, 2018, Art. no. 14.
[34] R. R. Muntz and J. C. S. Lui, "Performance analysis of disk arrays under failure," in *Proc. 16th Int. Conf. Very Large Data Bases*, 1990, pp. 162–173.
[35] M. Holland and G. A. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Proc. 5th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 1992, pp. 23–35.
[36] J. Wan, J. Wang, Q. Yang, and C. Xie, "S2-RAID: A new RAID architecture for fast data recovery," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–9.
[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, 2001.
[38] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
[39] A. Goel, C. Shahabi, S.-Y. D. Yao, and R. Zimmermann, "SCADDAR: An efficient randomized technique to reorganize continuous media blocks," in *Proc. 18th Int. Conf. Data Eng.*, 2002, pp. 473–482.
[40] 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 *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 20.
[41] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.
[42] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 81–94.

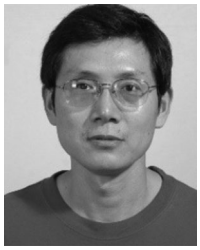
[43] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 213–226.
[44] Z. Li, M. Lv, Y. Xu, Y. Li, and L. Xu, "D3: Deterministic data distribution for efficient data reconstruction in erasure-coded distributed storage systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 545–556.

Liangliang Xu received the BS degree with the Department of Information and Computational Science, Anhui University, Hefei, China, in 2017. He is currently working toward the PhD degree in the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. His research interests include distributed storage systems, data recovery, and erasure coding.

Min Lyu received the BS and MS degrees in applied mathematics from Anhui University, Hefei, China, in 1999 and 2002, respectively, and the PhD degree in applied mathematics from the University of Science and Technology of China, Hefei, China, in 2005. She is currently an associate professor with the School of Computer Science and Technology, University of Science and Technology of China. Her research interests include distributed storage systems, social networks, and security and privacy.

Zhipeng Li received the BS and PhD degrees in computer science from the University of Science and Technology of China, Hefei, China, in 2013 and 2019, respectively. His research interests include storage systems, solid state devices, and erasure coding.

Yongkun Li received the BS degree in computer science from the University of Science and Technology of China, Hefei, China, in 2008, and the PhD degree in computer science and engineering from the Chinese University of Hong Kong, Hong Kong, in 2012. He is currently an associate professor with the School of Computer Science and Technology, University of Science and Technology of China. His research mainly focuses on data-intensive computing systems, with emphasis on file systems, and memory systems.



Yinlong Xu received the BS degree in mathematics from Peking University, Beijing, China, in 1983, and the MS and PhD degrees in computer science from the University of Science and Technology of China (USTC), Hefei, China, in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology, USTC. He served the Department of Computer Science and Technology, USTC as an assistant professor, a lecturer, and an associate professor. He is currently leading a group in doing some networking, storage, and high performance computing research. His current research interests include storage system, file system, social network, and high performance I/O. He was the recipient of the Excellent PhD Advisor Award of the Chinese Academy of Sciences, in 2006 and Baosteel Excellent Teacher Award, in 2014.