Geometric Partitioning: Explore the Boundary of Optimal Erasure Code Repair

Yingdi Shan
Tsinghua University

Kang Chen[*]
Tsinghua University

Tuoyu Gong
Tsinghua University

Lidong Zhou
Microsoft Research

Tai Zhou
Alibaba Group

Yongwei Wu
Tsinghua University

Abstract

Erasure coding is widely used in building reliable distributed object storage systems despite its high repair cost. Regenerating codes are a special class of erasure codes, which are proposed to minimize the amount of data needed for repair. In this paper, we assess how optimal repair can help to improve object storage systems, and we find that regenerating codes present unique challenges: regenerating codes repair at the granularity of chunks instead of bytes, and the choice of chunk size leads to the tension between streamed degraded read time and repair throughput.

To address this dilemma, we propose *Geometric Partitioning*, which partitions each object into a series of chunks with their sizes in a geometric sequence to obtain the benefits of both large and small chunk sizes. *Geometric Partitioning* helps regenerating codes to achieve 1.85× recovery performance of RS code while keeping degraded read time low.

CCS Concepts: **• Computer systems organization → Reliability**; **Redundancy**; **• Information systems → Distributed storage**; **Storage recovery strategies**.

Keywords: erasure coding, repair, object storage system

ACM Reference Format:
Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. 2021. Geometric Partitioning: Explore the Boundary of Optimal Erasure Code Repair. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483558

[*]Corresponding Author: Kang Chen (chenkang@tsinghua.edu.cn).

SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8709-5/21/10...$15.00
https://doi.org/10.1145/3477132.3483558

1 Introduction

Object storage systems such as Haystack [6], Amazon S3 [3] are widely used to store immutable Binary Large Objects (BLOBS), including photos, videos, archives, and documents. Erasure coding is widely used in such production systems (e.g., at Facebook [29], Microsoft [11, 20]) to achieve reliability at a reduced storage cost compared to replication. The amount of data needed for repair due to data loss is referred to as *repair cost*. Erasure codes, however, suffer from high repair cost, and there are many previous works trying to address this problem [8–10, 15, 18–20, 22, 25, 26, 28, 33, 37, 38, 41, 43, 44, 48]. Regenerating codes [15] are a special family of erasure codes designed to minimize the repair cost. Some of the regenerating codes, such as MSR (Minimum Storage Regenerating) codes, can achieve theoretically *optimal* repair cost while providing the same storage efficiency and reliability guarantees.

Recovery efficiency, which measures how fast the system can restore its original fault tolerance under certain hardware resource constraints by repairing from the available pieces to recover from data loss due to failures, and *degraded read time*, which measures the time to read an object in a degraded case when an object needs to be repaired as the server storing the object requested fails, are two important design metrics for erasure coding, and have been addressed by many previous works [13, 15, 20, 22, 26, 28, 29, 33, 37, 38].

Optimality in repair cost, however, does not necessarily improve recovery efficiency or reduce degraded read time in a practical object storage system using regenerating codes. Regenerating codes repair at the granularity of chunks [33, 37, 38], and each chunk is further divided into many small sub-chunks. These sub-chunks are correlated in a complex encoding structure. To repair a chunk, only a small portion of sub-chunks is read from corresponding disks. This dispersed disk access pattern causes fragmentation, leading to reduced disk performance and thus recovery efficiency.

The chunked repair granularity also leads to increased degraded read time. If we repair at the granularity of bytes like RS code (Reed-Solomon code), we can parallelize repairing and transferring repaired bytes to the client. However, for regenerating codes, we need to wait for the repair of the first chunk before transfer it back to the client, which can take as much time as repairing the object. Even worse, when reading an object whose size is smaller than the chunk size, unnecessary data is also repaired, which leads to read amplification and longer degraded read time.

When using regenerating codes, an object storage system often chooses a single chunk size as the unit for encoding [33, 38, 44]. Objects are then split into chunks before encoding. The choice of chunk size has fundamental implications on practical performance and must balance inherent trade-offs between degraded read time and recovery efficiency. A larger chunk size can help to reduce fragmentation and discontinuous reads for disk access, increasing recovery efficiency. But a large chunk size can increase the time to wait for the first repaired chunk and increase the chance of read amplification. As a result, the choice of chunk size becomes a dilemma.

Nevertheless, we find that low degraded read time and high recovery efficiency can be achieved *simultaneously*. The key is to use variable chunk sizes **inside** each object so that we can obtain the benefits of large and small chunks for each object, and optimize for both recovery efficiency and degraded read time. We start to repair from a small chunk size to avoid unnecessary waiting for the repairing of the first chunk, then we limit the ratio of adjacent chunk sizes so that the repair of the current chunk can predate the transfer of the previous chunk. Finally, we employ the largest possible chunk sizes under the above constraints.

Then we have *Geometric Partitioning*, which partitions each object into chunks with sizes in a geometric sequence(e.g., 4MB, 8MB, 16MB, 32MB, 64MB, and so on), and group chunks from different objects into buckets for encoding. We show that *Geometric Partitioning* resolves the inherent tension between recovery efficiency and degraded read time. The use of larger chunk sizes later in the sequence achieves high recovery efficiency with efficient continuous sequential reads. The use of earlier small chunk sizes helps to lower the degraded read time. We pack the fronts of objects, which are smaller than the smallest chunk size, but use RS code rather than regenerating codes to eliminate read amplification as they only consume a small portion of total space.

In summary, we make the following contributions:

- We point out the gap between theory and practice in the use of regenerating codes and show the inherent tension between degraded read time and recovery efficiency for regenerating codes when used in a large-scale storage system.
- We propose *Geometric Partitioning*, a novel method that introduces variable chunk sizes inside each object, elegantly resolves the tension.
- We build *RCStor*, a storage system for storing immutable objects, which is specially optimized for regenerating codes. We show that with *Geometric Partitioning* regenerating codes are efficient enough to be used in object storage systems: RCStor improves the recovery performance of Clay code to 1.30× of the recovery performance of LRC and 1.85× the recovery performance of RS code.

2 Background

2.1 Repair, Degraded Read and Recovery

In this paper, we use the term *repair* to describe the following procedure: collect necessary data from other nodes and decode the collected data to obtain original unavailable data.

For an object storage system, there are two important operations in relation to repair : (i) *degraded reads* to temporarily unavailable objects (e.g., system maintenance, network failures), (ii) *recovery* of a crashed disk or a failed node. They can occur hourly to daily, depending on the size of the cluster [42].

The time of degraded read, which can affect user experience and service's SLO (Service Level Objective), is an important metrics for object storage system [11, 13, 26, 28]. To read a temporarily unavailable object, a user issues an HTTP request to the object storage system, the HTTP server then repairs the object and transfer the object back to the client. It is obvious that repairing and transferring can be pipelined and conducted in parallel with each other. So degraded read time depends on the time to repair, the time to send the repaired object back to the client, and how they are pipelined. It should be noted that the network edge bandwidth is limited (e.g. 1Gbps) because there are often many clients competing for limited total edge bandwidth. Thus the total degraded read time can be easily dominated by transfer time plus the repair time of the first chunk.

With regard to recovery, efficient recovery can reduce MTTL (Mean Time to Loss), increasing the durability of the system [12, 20]. Efficient recovery can also reduce the number of degraded reads, lightening the burden of the system. Unlike degraded read time, which is mainly determined by latency (the time to repair the first byte), recovery efficiency is determined by throughput. For modern data centers where high-speed network is common [2], disk bandwidth is becoming the determinant factor for recovery throughput, since disk bandwidth is more difficult to be fully utilized.

2.2 Repair for Different Codes

A replication-based system can repair simply by copying as in Figure 1(a). But erasure-coded systems need to repair the lost data and is much more complicated. A (k, r) code generates r parities from k data nodes. For a fair comparison, we set all erasure codes with $k = 10$ data disks and $r = 4$ parity disks, which is a common practice in production systems [29, 44]. Denote there are 10 data nodes from D_1 to D_{10}, 4 parity nodes from P_1 to P_4. An illustration of repair procedure from a single node failure is given in Figure 1. The quantitative comparison of these codes is in Table 1. We

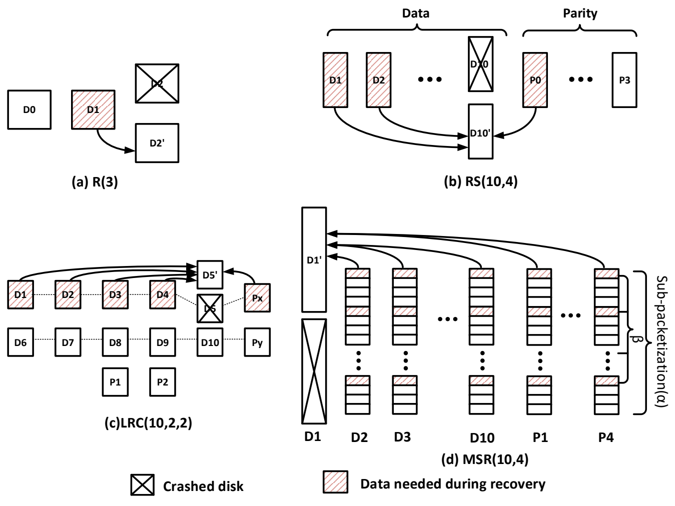


Figure 1. Repair patterns for different methods.

mainly focus on single-node failures because it is the dominating (> 98%) case in practice [22, 42]. It's still important to repair from multi-nodes failures for reliability, but we care less about its efficiency.

Repair for RS code. RS code is a common and widely used erasure code [40]. Figure 1(b) shows the repair of RS code. To repair from a single-node failure, we need to gather **all** data from k survived nodes. Then the lost data will be repaired by decoding or re-encoding. The process is time-consuming and involves significant disk I/O and network traffic. The reliability of RS code is high because RS code is an MDS (Maximum Distance Separable) code. MDS codes are codes that can repair from the failure of *any* combinations of r nodes.

Table 1. Codes Comparison. Read traffic represents the ratio of disk/network traffic to the amount of data repaired.

	Storage	MDS	Read traffic	Sub-packetization
RS(10,4)	140%	Yes	10	1
LRC(10,2,2)	140%	No	5.71 [1]	1
Clay(10,4)	140%	Yes	3.25	256

Repair for LRC codes. LRC [20] (Local Reconstruction Codes) reduces the network traffic and disk I/O during repair by connecting to fewer nodes. Figure 1(c) shows the repair of LRC codes, where P_x and P_y are two local parity nodes. LRC codes divide nodes into multiple groups, generate local parities for each group, and then generate global parities to improve reliability. When a node fails, only nodes from the same group need to be accessed, thereby reducing both disk I/O and network traffic. However, LRC codes do not have the same storage-reliability trade-off as RS code because LRC codes are not MDS codes. LRC codes cannot tolerate all failure node combinations, weakening their reliability.

Repair for Regenerating codes. Unlike LRC codes that reduce I/O by connecting to fewer nodes, regenerating codes need to read from d ($d > k$) nodes. Regenerating codes reduce I/O by introducing finer-granularity sub-chunks and more complex connections between them.

The repair pattern of regenerating codes is shown in Figure 1(d). Regenerating codes organize a data chunk into multiple sub-chunks. When a node fails, only a small number of sub-chunks from d nodes are needed during data repair. The number of sub-chunks stored in each node is called sub-packetization denoted as α. The number of sub-chunks required for each node during repairing is denoted as β. These are two critical parameters for regenerating codes and will influence the design of regenerating coded object storage systems.

In general, a small value for α or β is preferred because it can provide better locality. For example, given $\alpha = 64, \beta = 16$, there will be 16 discontinuous sub-chunks in each node during repair in the worst case. Reducing the parameters to $\alpha = 16, \beta = 4$, the disk I/O remains the same (the data needed during repair kept the same), but the number of discontinuous sub-chunks in the worst case can be reduced to 4.

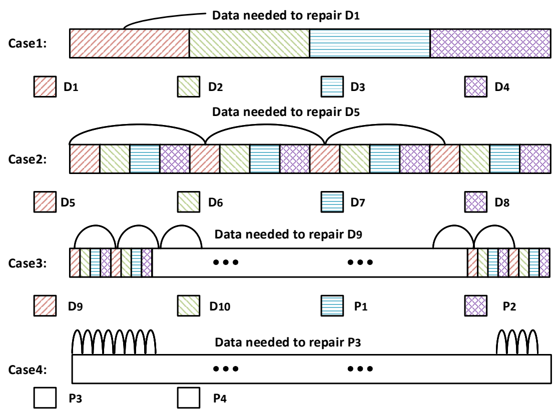


Figure 2. Repair patterns of a chunk for Clay(10,4) code when different disks fail. Case 1 shows sub-chunks that need to be read when D1-D4 fails, Case 2 shows sub-chunks for D5-D8, and so on. Each colored block represents 64 sub-chunks for Case 1, 16 sub-chunks for Case 2, 4 sub-chunks for Case 3, and 1 sub-chunk for Case 4. When a disk failed, sub-chunks with the same color need to be read for repair.

[1] Two global parities need to read 10x of data to repair, while other nodes only need to read 5x of data for repair. So the average read traffic of LRC is $\frac{10\times2+5\times12}{14} \approx 5.71$.

There are various kinds of regenerating codes, including MSR codes [17, 25, 33, 37, 43, 44], MBR (Minimum Bandwidth Regenerating) codes [39], Hitchhiker code [38], Simple Regenerating codes [34], etc. Among them, MSR codes are the most attractive because they keep the same storage-reliability trade-off as RS code while providing *optimal* repair cost.

In this paper, we mainly focus on a recently constructed MSR code, Clay code [44]. Clay code has many nice properties including flexible coding parameter settings, less computing complexity. Clay code is one of the regenerating codes that require the minimum possible amount of data to be read from disk during a repair, and it is proved that Clay code has a minimum possible α among these codes [5]. Clay code also provides a good locality to be used in a real-world storage system. The rest of the paper will mainly use Clay code by default, but our method is suitable for all regenerating codes.

Figure 2 shows the recovery pattern for a data chunk in Clay(10,4) code, where $d = 13$, $\alpha = 256$ and $\beta = 64$. The 256 sub-chunks on each disk can be indexed into four quaternary digits (s_0, s_1, s_2, s_3). When D_i (P_i can be seen as D_{k+i}) fails, we need to read all sub-chunks that satisfy

$$s_{\lfloor \frac{i-1}{r} \rfloor} \equiv i - 1 \pmod r$$

from the remaining 13 disks. In short, we should read $\frac{\beta}{\alpha} = \frac{1}{4}$ amount of data from each survived node.

3 Challenges for Applying Regenerating Codes

In this section, we study the challenges of applying regenerating codes in an object storage system, and why low degraded read time and high recovery efficiency are unattainable *simultaneously* in storage systems.

3.1 The Effect of Chunk Size

Regenerating codes are used together with the sub-chunking (also called hop-and-couple) technique to reduce disk I/O [33, 37, 38]. Data is organized into chunks before encoding, and regenerating codes repair data at the granularity of *chunk*. This means that you cannot encode using a large chunk size and decode using a smaller chunk size without incurring read amplification like RS code. Such phenomenon makes the choice of chunk size challenging.

Large Chunk Size Benefits Recovery. Though regenerating codes greatly reduce the amount of data to be read, they introduce fragmentation and discontinuous reads. Take case 3 from Figure 2 as an example. To repair a chunk, you need to read 64 sub-chunks, which are 16 discontinuous reads, and the I/O size of each read is the size of 4 sub-chunks. If the I/O size is 4KB and the size of sub-chunk is 1KB, the corresponding *chunk size* will be $4KB \times 64 = 256KB$. For case 4, the corresponding chunk size is as large as 1MB. Any chunk size smaller than that will result in reduced performance.

The requirement for chunk size is much higher to fully utilize the bandwidth of HDD. For an HDD, the I/O size needs to be as large as 4MB (the corresponding chunk size is 256MB for case 3) or even 8MB to amortize I/O latency and utilize disk bandwidth better [30].

Overly Large Chunk Size Harms Degraded Read. However, it's infeasible to increase *chunk size* indefinitely, because a *chunk size* larger than object size can cause read amplification and increase degraded read time. An object storage system contains objects of various sizes, from several KBs to multiple GBs. If we choose a 256MB *chunk size*, we need to repair the whole 256MB chunk only to read a 64MB object in that chunk. In fact, degraded read requests whose sizes are smaller than chunk size can lead to additional disk read. A smaller chunk size can reduce such phenomenon.

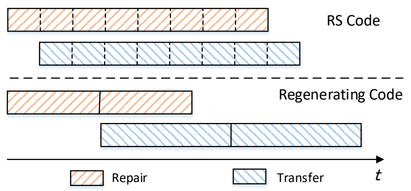


Figure 3. An illustration of degraded read time of RS Code and Regenerating Codes.

Small Chunk Size Benefits Degraded Read. The process of degraded read can be divided into two steps: repair the necessary data from storage servers and transfer the repaired object from the server to end-users. For large objects, by dividing the process into multiple small steps, read time can be greatly reduced by pipelining as in Figure 3. This is natural for RS code. However, for regenerating codes, the whole process will be blocked by the repair of the first chunk when chunk size is large. For example, most objects can not be pipelined when chunk size is as large as 256MB. A small

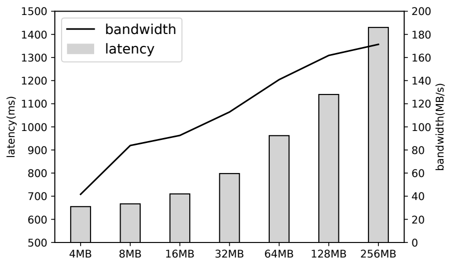


Figure 4. Average degraded read time to read a 64MB object at different chunk sizes, together with the corresponding average utilized disk read bandwidth for recovery using Clay(10,4) code on a HDD.

chunk size can reduce blocking time and benefit degraded read.

Conclusion. Figure 4 shows the trade-off between degraded read time and recovery efficiency at different chunk sizes using Clay(10,4) code. We use the average disk read bandwidth (the harmonic mean of 4 cases in Figure 2) to measure recovery efficiency because it can measure the effect of discontinuous read quantitatively. When chunk size is larger than object size, the extra repaired data is discarded. The degraded read time for a client to read a 64MB object over a 1Gbps network increases from 700ms to over 1,300ms when we increase chunk size from 4MB to 256MB, but the disk read bandwidth increases from 40MB/s to over 170MB/s.

Our study shows a fundamental trade-off between degraded read time and recovery efficiency, where *chunk size* plays a key factor. A large chunk size can improve recovery efficiency, at the cost of more severe read amplification and increased blocking time, leading to longer degraded read time, and vice versa.

3.2 Case Study: Applying Regenerating Code into Existing Data Layouts

In this section, we study two common data layouts in Figure 5, and show how degraded read time and recovery efficiency will be affected by the introduction of regenerating code.

Contiguous Layout. Contiguous layout packs multiple objects together to form equal-sized large files, which are encoded together with each file as a whole. As a result, parity is generated from data in **different** objects. A normal read to Contiguous layout only goes through a single disk. Facebook f4 [29], for example, adopts Contiguous layout.

To apply regenerating codes into Contiguous layout, an arbitrary large chunk size can be selected to make recovery highly efficient. However, Contiguous layout can have severe read amplification because objects are not aligned to each encoding chunk and several objects can be packed into a single chunk. Thus a degraded read of a small object may span to a whole or multiple chunks, leading to increased degraded read time.

Stripe Layout. Stripe layout is widely used by systems such as HDFS 3.0 [16] and QFS [32]. Stripe layout splits each object into small stripes, spreads each stripe into multiple nodes, and generates the corresponding parities as Figure 5 (b) illustrates. A read request is processed by invoking k small requests and merging responses.

For Stripe layout, *chunk size* can be set as *strip size* (also called as stripe depth) so that degraded read requests to different stripes can be pipelined. To increase recovery efficiency, chunk size can be set to as large as $\frac{1}{k}$ of the object size at the cost of less efficient pipelining. Any chunk size larger than that will cause read amplification. Thus, the chunk size of

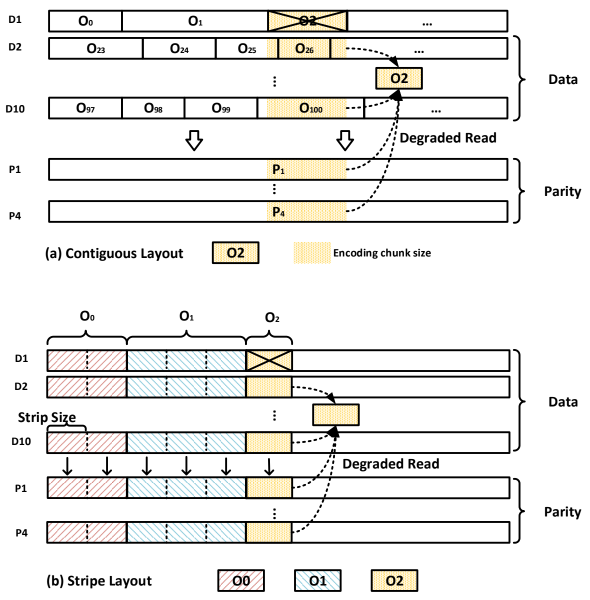


Figure 5. Contiguous layout and Stripe layout.

a 100MB object is limited to 10MB, and the corresponding sub-chunk can be as small as 40KB. We can find from Figure 4 that such a small chunk size is far from fully utilizing disk bandwidth.

In summary, we find that none of the existing layouts can achieve low degraded read time and high recovery efficiency *simultaneously* when regenerating codes are applied.

4 Geometric Partitioning

From the previous analysis, we can conclude that a fixed chunk size cannot utilize regenerating codes well. A natural consideration is to use different chunk sizes for different objects. Yet this solution still faces the same dilemma: pipelining will be less efficient if a large chunk is selected, recovery will be less efficient if a small chunk is selected.

The design of *Geometric Partitioning* is based on the following principle: if a **single** object can be partitioned into chunks with **variable** sizes, then we can then use smaller chunks to reduce degraded read time through pipelining and use larger chunks to achieve efficient sequential reads, then we can obtain the benefits of both small and large chunk sizes.

Figure 6 illustrates the overall design of geometric partitioning, which partitions an object into multiple chunks, each to its corresponding **bucket**. All chunks of an object are put into a single disk. Each bucket is a large file on a disk, containing equal-sized chunks from different objects. The sizes of these buckets form a geometric sequence. Buckets from $k + r$ disks are encoded together using regenerating code.

There are two predefined parameters for *Geometric Partitioning*, s_0 and q, where s_0 is the initial value of the geometric sequence and q is the common ratio of the sequence. An object with size S can thus be represented by the combination of the sequence, such that $S = R + \sum_{i=1}^{n} a_i \cdot s_0 q^{i-1}$, where $R = S \mod s_0$, a_i is the number of chunks that fall into a specific bucket, and n is the number of chunks that the object is partitioned into.

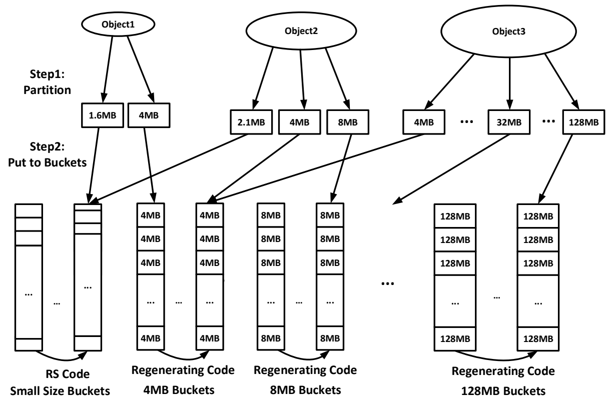

Object1
Object2
Object3
Step1: Partition
1.6MB
4MB
2.1MB
4MB
8MB
4MB
32MB
128MB
Step2: Put to Buckets
RS Code Small Size Buckets
Regenerating Code 4MB Buckets
Regenerating Code 8MB Buckets
Regenerating Code 128MB Buckets

Figure 6. An illustration of *Geometric Partitioning*.

The design of *Geometric Partitioning* is detailed below.

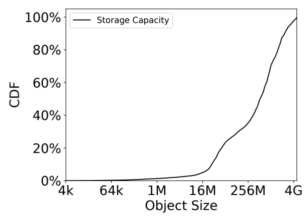
4.1 Cut Front to Eliminate Read Amplification

Section 3.1 tells us that read amplification is caused by read requests whose sizes are smaller than the chunk size. Section 3.2 tells us that such requests are brought by the lack of object alignment. A naive way to align objects is to encode objects of the same size. However, it is hard to find enough objects (e.g., 10 for a Clay(10,4) code) of the same size.

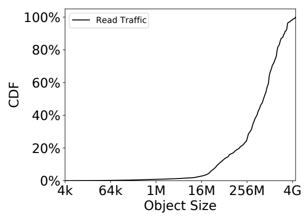
A smarter way is to cut the front of each object. For each object, we subtract a portion of it such that the remaining size of this object is a multiple of s_0 and put that portion into a separate bucket, called small-size-bucket. Since the remaining size of the object is a multiple of s_0, as long as s_0 is large enough (e.g., 4MB), it is much easier to find a chunk with the same size. Read amplification can thus be eliminated for the remaining part of the object.

Small-size-buckets are used to store the front portion of objects. Objects smaller than s_0 are also placed directly into small-size-buckets. Unlike other buckets, there is no specific bucket size for small-size-buckets, and the size of different objects in a small-size-bucket can vary. Small-size-buckets are encoded by RS code so that read amplification in small-size-buckets can be eliminated. In addition, the presence of RS-coded small-size-buckets will allow the pipeline of degraded reads to begin almost immediately, reducing the time for degraded reads.

Based on our investigation of a production trace in Figure 7 we find that storage capacity is dominated by larger



(a) Byte CDF of Capacity



(b) Byte CDF of Read Trafic

Figure 7. Trace from Alibaba Cloud Object Storage Service which contains videos, images, photos, documents, archives, etc. The trace is available at https://github.com/rcstor/ali-trace.

objects (> 97.7% capacity is consumed by objects larger than 4MB). Traces from Facebook [36] and Microsoft [11] also support this observation.

The storage capacity consumed by small-size-bucket is small when a proper s_0 is selected, indicating that the disk and network traffic incurred by the recovery of small-size-bucket is small. This implies that the existence of small-size-bucket has a limited impact on recovery efficiency.

4.2 Partition Objects Geometrically to Benefit Recovery and Degraded Read

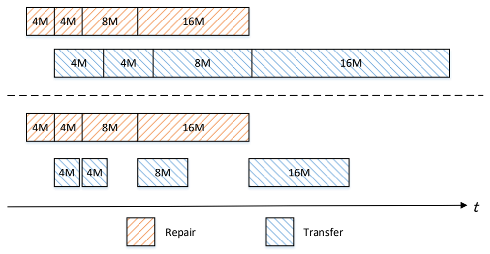


Figure 8. Two potential cases for pipelining in *Geometric Partitioning*.

To improve recovery efficiency, we should increase chunk sizes. Theoretically, the best way to partition an object is to not partition, so that chunk sizes are maximal. However, without partitioning, degraded read time on large objects will suffer due to the lack of pipelining.

The introduction of variable chunk sizes helps to do pipelining more efficiently while putting most bytes of an object into larger chunks. The main ideas are as follows: (i) start to repair from a small chunk size to avoid unnecessary waiting for the repairing of the first chunk, (ii) the ratio of adjacent chunk sizes (i.e. $\frac{s_i}{s_{i-1}}$) should be limited so that the repair of the current chunk can predate the transfer of the previous chunk as in Figure 8, (iii) employ largest possible chunk sizes under the above constraints. The above three principles inspire us that chunk sizes should grow exponentially, which leads to the design of *Geometric Partitioning*.

Geometric Partitioning partitions front-cut-objects into chunks with sizes forming a geometric series, starting from s_0. An illustration of it is given in Figure 8. For example, a 32MB object can be partitioned into four chunks, 4MB, 4MB, 8MB, and 16MB. To repair the object, the first 4MB chunk is sent to the user as soon as it is reconstructed. The regeneration of subsequent chunks can be processed together with the transfer of the previous chunk. From Figure 8, we can see that degraded read time is close to transfer time when transfer is less blocked by the repairing of chunk. When transfer is blocked by repair, though not optimal, the pipelining of *Geometric Partitioning* still helps to reduce degraded read time.

Besides the benefit of pipelining, geometric bucket sizes also allow large objects to put most of their data in buckets with large chunk sizes, resulting in better efficiency. By using a geometric sequence, instead of an arithmetic sequence or a constant sequence, we can limit the number of partitioned chunks to the logarithm of the object size, rather than linear or polynomial to the object size. This can help to increase the average chunk size.

4.3 Help Pipelining by 2-pass Scan

To better utilize pipelining and reduce blocking time, we need to make sure the coefficient of each chunk, a_i, is non-zero. For instance, assuming $s_0 = 4MB$ and $q = 2$, if we partition a 20MB object into 4MB+16MB chunks, these 2 chunks may not be properly pipelined since their size gap is large, resulting in a slow read. If the coefficient of each chunk is non-zero, the size gap between adjacent chunks will be limited, resulting in a situation similar to Figure 8. The coefficients can be found by Algorithm 1.

Algorithm 1 scans the geometric sequence twice to determine the coefficients. The first scan subtracts the size of **every** bucket until the remaining size is too small to be filled into a bucket. The subsequent scan uses a greedy policy, trying to choose the largest possible chunk size until no bucket can be filled.

For example, suppose the size of an object is 73.5MB. The first pass will split the object as 4MB + 8MB + 16MB + 32MB. And the remaining size will be split as 8MB + 4MB + 1.5MB in the second pass. Thus, the final result will be 73.5MB = 1.5MB + 2×4MB + 2×8MB + 16MB + 32MB.

4.4 Parameter Setting

Geometric Partitioning has two parameters to tune. A larger s_0 can enlarge chunk sizes, thereby increasing average disk read bandwidth, at the cost of increased disk/network traffic due to RS code. A larger s_0 may also add overhead to pipelining since the first chunk cannot be pipelined, increasing degraded read time. With respect to the selection of q, a small q can help repair predate transfer easier for pipelining. But a small q may not be optimal for recovery efficiency. So both s_0 and q should be tuned through the sampling of target workloads to achieve balanced performance between degraded read latency and recovery efficiency. Grid search can be used to collect performance data and help set parameters for preferred characteristics (better recovery or better degraded read time).

Algorithm 1: Geometric Partitioning

```
Data: The object size $S$.
Result: $R$, $n$, $a = (a_1, a_2, \cdots, a_n)$
begin
    $i \leftarrow 1$;
    while $S \geq s_0 \cdot q^{i-1}$ do
        $a_i \leftarrow 1$;
        $S \leftarrow S - s_0 \cdot q^{i-1}$;
        $i \leftarrow i + 1$;
    $n \leftarrow i$;
    while $i \geq 1$ do
        while $S \geq s_0 \cdot q^{i-1}$ do
            $S \leftarrow S - s_0 \cdot q^{i-1}$;
            $a_i \leftarrow a_i + 1$;
        $i \leftarrow i - 1$;
    $R = S$;
```

5 System Design and Implementation

5.1 System Design

Architecture. We have designed and implemented *RC-Stor*, a storage system for storing immutable objects, to show the effectiveness of *Geometric Partitioning*. The system contains 3 roles including *Directory Server*, *Storage Server* and *HTTP Server*. *HTTP Server*s are responsible to handle user requests. *Directory Server*s store meta information and monitor the whole system. *Storage Server*s store and manage the objects and also store index files to track objects. Each *Storage Server* is bound to a specific disk. Multiple processes of *Storage Server*s can run on the same machine with multiple disks. Recovery tasks are dispatched by *Directory Server* to *Storage Server*s. The reliability of *Directory Server* can be achieved by using a replicated state machine [31].

Placement Groups. To utilize the bandwidth of more disks and NICs for recovery, we use a similar concept of PG (Placement Group) as in Ceph [45]. Our system contains thousands of PGs, where each PG is a group of $k + r = 14$ disks on 14 different nodes, representing the set of disks to be encoded. The information about PGs is stored on *Directory Servers*. Each disk can belong to multiple PGs and each PG can recover independently. When one disk fails, all PGs that are related to the failed disk will begin to recover. So instead of using only $d = 13$ disks to recover, there will be more disks utilized to accelerate recovery. The *Direcotory server* assigns disks to PG so that the maximal amount of disks are correlated to recovery when a disk fails. An object is first mapped to a PG based on the hash of its ID. Then, the disk with the least capacity consumed for that PG is selected to put the object. Finally, the object is partitioned into chunks and put into its corresponding buckets.

Metadata Management. For each PG, $r + 1$ out of $r + k$ disks are randomly selected to store the replicated index of that PG. Each index file tracks the metadata of all objects in that PG and is loaded into memory when the *Strorage Server* restarts. Each record of the index file includes object ID, size, disk ID, checksum, and positions of partitioned chunks of the object. Since all chunks in a bucket are aligned, the position of a chunk can be stored within 2 bytes except for the small-size-bucket. As a result, the average metadata size of an object is about 40 bytes, which is small enough to be kept in memory.

Put and Get. RCStor provides streaming HTTP interfaces to end-users by specifying HTTP Content-Type as application/octet-stream.

To put an object, we first put it into a replication-based object storage system. Then, we export these objects from the replication-based storage system into the erasure-codes-based storage system using background processes. This design is similar to Facebook F4 [29]. By exporting these objects in batch, we can avoid the costly overhead of parity updating.

To get an object, *HTTP Server* fetches the index from that PG's corresponding *Storage Server*. If it is a degraded read request (the corresponding disk is offline or error occurred during normal read), *HTTP Server* will collect the necessary data from *Storage Servers*, regenerate and transfer (which is pipelined) that object to the client. Otherwise, the object will be transferred to the client directly (which is also pipelined).

Although Geometric Partitioning requires reading from multiple locations rather than reading from a contiguous area, the performance penalty for normal reads can be small (as evaluated in Section 6.2) because the chunk size is large enough to amortize the extra time consumed on disk seeks.

IO Scheduling. On every *Storage Server*, there is a FIFO request queue bound to multiple threads that pull from it continuously. Each normal or degraded read request contains the offsets and sizes of all correlated chunks of an object, so that request does not have to be queued more than once for different chunks of a single object. To handle a request, *Storage Server* will read all correlated data into 256KB (or smaller) packets and push them to *HTTP Server*. Background requests (recovery, data importing, etc.) to *Storage Server* are put into separate queues and processed in a lower priority.

Paralleled Recovery. To recover a node, *RCStor* do not recover at the granularity of objects, but at the granularity of chunks. The chunks of an individual object can be recovered concurrently. There is a global queue on *Directory Server*, managing all the chunks that are pending recovery. Each recovery task contains the position information of a single chunk. *HTTP Servers* pull these tasks from *Directory Server* continuously and recover these chunks concurrently. Recovery tasks are weighted by their chunk sizes (e.g. 1 for 4MB chunk, 64 for 256MB chunk), and there is a global weight representing the maximal possible total chunks size of concurrent repairing chunks (e.g. If global weight is 512, we can recover eight 256MB chunks concurrently, or we can recover four 256MB chunks and eight 128MB chunks concurrently).

5.2 Implementation

We have implemented *RCStor* using 10,694 lines of Golang code. There are some implementation details as below:

Encoding Optimization. We implement Clay code using C. Dedicated optimization is made to minimize the time encoding/decoding consumed. Standard optimizations such as SIMD instructions[35] and loop unrolling are used to improve the performance. We also exploit software prefetching, streaming write instructions to reduce the overhead of memory access. We have achieved 22.3GB/s for encoding, 18.5GB/s for decoding, and 5.0GB/s for regenerating using multiple threads on a 12-core server for Clay(10,4) code.

Memory Management. There is a memory pool for HTTP Server to manage repaired chunks. Each repaired chunk can remain in memory for a certain amount of time (should be decided by the memory of the server), after that the repaired chunk is flushed to disk. Further requests to that chunk are redirected to disk. We can prevent slow clients from consuming too much memory in this way. Furthermore, we only allow allocating chunks smaller or equal to 256MB to avoid unnecessary large chunks.

Range Access Support. In some cases, the system needs to support range access where users can download a portion of a large object by specifying offset and length. *RCStor* supports ranged degraded read by reading from the first related chunk and discarding unnecessary data.

6 Evaluation

6.1 Evaluation Setup

Hardware Setup. We use a cluster of 16 servers, each equipped with dual Intel Xeon E5 2643 v4 CPU, 128GB 2133 MHZ DDR4 RAM, a 512GB SATA3 SSD, and 6× 8TB 7200rpm SAS HDD. All servers are running CentOS 7.8. Servers are connected via a 56Gbps Infiniband network running IPoIB. All servers are acting as both an HTTP server and a Storage Server. We choose three servers to act as the Directory Server. XFS is used as the underlying file system, and all file system cache is cleared at the beginning of all experiments.

Table 2. Description of workloads.

Workload	Object Size	Avg Object Size	# Objects	Avg Request Size	Hardware	Total Capacity	Capacity per Disk
W_1	4MB~4GB	102.8MB	170000	148.5MB	16×6 HDDs	24.4TB	255GB
W_2	4KB~4MB	101.3KB	500000	72.0KB	16×1 SSDs	70.9GB	4.4GB

We run clients on the same machine running servers. Each machine runs at most 8 clients. We set the network bandwidth of each client to 1Gbps. This is a common bandwidth for network edge [26]. We also evaluate how *Geometric Partitioning* performs when client bandwidth varies.

Workloads. We evaluate our system on HDD and SSD respectively to show the generality of *Geometric Partitioning*. We evaluate two workloads, both sampled from the trace in Figure 7. W_1 contains objects from 4MB to 4GB, representing a workload with larger objects (e.g., archives, docker images, videos), and we use 16×6 HDDs to store them. W_2 contains objects from 4KB to 4MB, representing a workload with smaller objects (e.g., photos, documents), and we use 16×1 SSDs to store them. The detailed description of the two workloads is in Table 2. Production system like Giza [11] employs a similar method to separate objects into different systems.

W_1 reveals how *Geometric Partitioning* perform for large objects on HDDs, W_2 reveals how *Geometric Partitioning* perform for small objects on SSDs. *Geometric Partitioning* is not suitable for storing large objects on SSDs or storing small objects on HDDs, as naive solutions like striping or using RS code can provide comparable performance.

Parameter Settings. For W_1, we set s_0 = 1MB, 4MB, 16MB and q = 2 for *Geometric Partitioning*, strip size as 256KB for Stripe layout, and we choose three chunk sizes: 16MB, 64MB and 256MB for Contiguous layout. For W_2, we set s_0 = 128KB, 256K and q = 2 for *Geometric Partitioning*, strip size as 32KB for Stripe layout, and we choose two chunk sizes: 128KB and 512KB for Contiguous layout.

We use parameter settings (10,4) for Clay code and RS code, (10,2,2) for LRC code. We also add Stripe-Max as a comparison. Stripe-Max is a special type of Stripe layout where the strip size of each object is the size of that object divided by k. We also evaluate ECPipe [26], a pipelining technique designed to reduce network congestion. We use RS code as its underlying code because ECPipe only applies to erasure codes with addition associativity, which does not hold for Clay code. We set the packet size of ECPipe the same as the strip size of Stripe layout.

We set the global weight as 512 so that each HTTP server can recover up to 512 chunks concurrently. We have 16 servers, so we can recover up to 8192 chunks at the same time.

In all the following experiments for different layouts, Clay code is used. And for RS code and LRC code, we choose Stripe layout, because the three challenges solved by *Geometric Partitioning* do not exist for RS code and LRC code, and the use of Stripe layout helps these codes get the best performance. We also evaluate Hitchhiker code, a non-optimal regenerating code, as a baseline. We use *Geometric Partitioning* as its layout. We set s_0 = 4MB for W_1, s_0 = 128KB for W_2, and set q = 2 for Hitchhiker code.

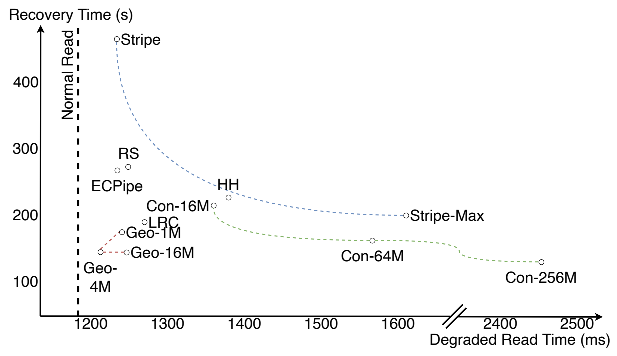
Implementation Optimizations for Baselines. For the recovery of Stripe/RS/LRC, instead of recovering the whole disk stripe by stripe, I/O requests are batched to 4MB data blocks (we find that 4MB block size can maximize performance) to reduce the cost of synchronization and software overhead. It is worth noting that, for stripe layout, due to the inherent property (sub-chunking) of regenerating codes, the scattered disk read pattern remains unchanged when this optimization is applied as the underlying data layout remains unchanged.

To further reduce the degraded read time of Stripe/RS/LRC, we send n instead of k requests to storage servers and begin to rebuild once receiving the first k responses for Stripe/RS. We may need to receive the first $k + 1$ responses for LRC because LRC codes are not MDS codes. This optimization makes degraded reads of Stripe/RS/LRC resilient to background load imbalance and unforeseen stragglers.

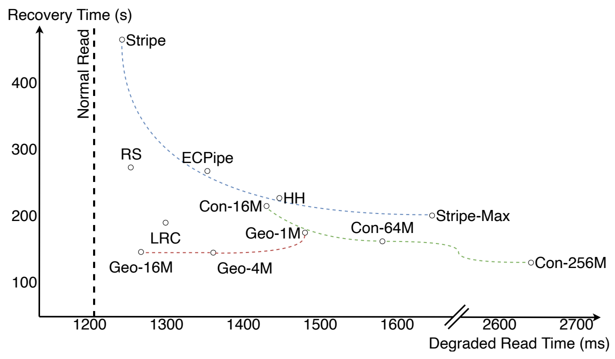
6.2 Evaluation on Production Trace

Methodology. We evaluate recovery efficiency by turning off one disk, starting recovery manually, and measuring the time to complete recovery. We recover all the failed PGs concurrently to measure the maximal recovery throughput of our system. Though we only turn off one disk, we are testing the average recovery time of all 4 cases in Figure 2 because there are multiple PGs on a single disk.

We also evaluate the average degraded read time before *RCStor* begin to recover. We send requests to unavailable objects following the distribution in Figure 7 and measure the time to get the last byte of these objects. We exclude degraded read requests to the same object to eliminate the effect of page cache. For Stripe, Stipe-Max, RS, LRC, and EC-Pipe, though only a part of an object is unavailable, we evaluate the degraded read time to read the whole object. For *Geometric Partitioning*, Hitchhiker, and Contiguous layout, we only evaluate the degraded read time of fully unavailable objects. Results are evaluated when the system is idle and when there are foreground workloads respectively. We use 15 nodes, 8 clients on each node, making requests continuously

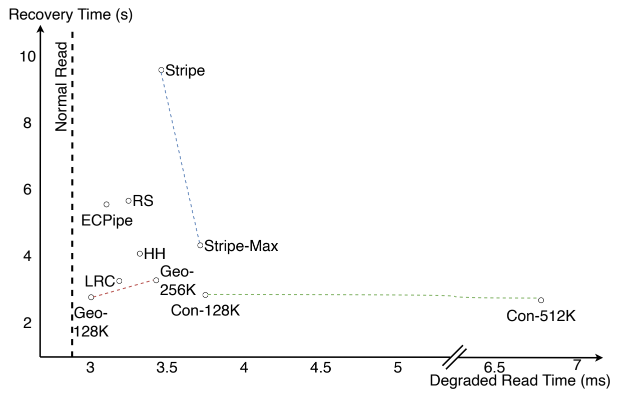


(a) Idle

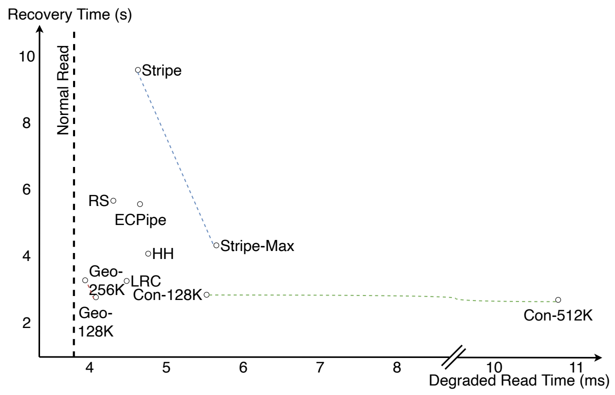


(b) Busy

Figure 9. Recovery time and degraded read time for W_1. Geo stands for *Geometric Partitioning*, Con stands for Contiguous layout, and HH stands for Hitchhiker code.



(a) Idle



(b) Busy

Figure 10. Recovery time and degraded read time for W_2.

following the distribution in Figure 7 to simulate the foreground workload. The disk bandwidth fluctuates between ~30MB/s to ~100MB/s for W_1, fluctuates between ~30MB/s to ~70MB/s for W_2 under this foreground workload.

Normal Reads. We test the average time of normal read requests. Our results show that Contiguous layout, Stripe layout and *Geometric Partitioning* have close normal read time ($1190ms \pm 20ms$ for W_1, $2.8ms \pm 0.2ms$ for W_2 when the system is idle; $1220ms \pm 30ms$ for W_1, $3.7ms \pm 0.3ms$ for W_2 when the system is busy). We draw the mean normal read time for these layouts in Figure 9 and Figure 10.

Degraded Read Time and Recovery Time. In Figure 9 and Figure 10, we compare the recovery time and degraded read time for different layouts, different parameter settings and different erasure codes on W_1 and W_2 respectively. All layouts and codes in Figure 9 and Figure 10 occupy similar amount of storage capacity (±%1). The result of the same layout and different parameter settings are connected by dotted lines. We vary s_0 for *Geometric Partitioning*, vary chunk size for Contiguous layout, and vary strip size for Stripe layout.

The recovery performance of Contiguous layout depends on the selection of chunk size. With a large enough chunk size, Contiguous layout can have a better recovery performance than *Geometric Partitioning*, at the cost of much longer degraded read time.

The performance of Stripe layout also depends on the selection of chunk size. With chunk size is as small as 256KB, the recovery performance is unacceptable. When we select the maximal possible chunk size as in Stripe-Max, recovery is much better, at the cost of increased degraded read time. Even so, Stripe-Max still costs about 1.37× of time to recovery Geo-4M for W_1.

By combining with *Geometric Partitioning*, the recovery of Clay code achieves 1.73GB/s for W_1, which is 1.85× compared to RS code, 1.30× compared to LRC. And the average degraded read time is only 1.02x normal read time. Without

Geometric Partitioning, Clay code cannot have better recovery performance than LRC code with acceptable degraded read time. For W_2, Clay code achieves 1.53GB/s, which is 2.01× compared to RS code. Though *Geometric Partitioning* reads more data during degraded read than normal read, their time is close because the repair time is hidden by transfer time through pipelining.

We read the same amount of data for RS and LRC because they all use Striping. However, LRC codes may need more requests because LRC codes are not MDS codes, so the degraded read time of LRC can be longer than RS code.

We've also varied s_0 for *Geometric Partitioning*. The results indicate that the choice of s_0 can affect the trade-off between degraded read time and recovery efficiency. When the system is idle, selecting a smaller s_0 can help pipelining and reduce degraded read time. However, when the system is busy, a larger s_0 can reduce the number of chunks to be read, thus reduce the possibility of I/O contention and the read time.

6.3 Performance Breakdown

Size of Small-size-buckets. Small-size-buckets for *Geometric Partitioning* occupies 1.7%, 3.7%, 9.4% of the total storage capacity for s_0 = 1MB, 4MB, 16MB respectively on W_1. Small-size-buckets for *Geometric Partitioning* occupies 26.7%, 35.4% of the total storage capacity for s_0 = 128KB, 256KB respectively on W_2.

Average Chunk Sizes. Though a larger average chunk size does not necessarily imply more efficient recovery (two different workloads with the same average chunk size can have different recovery efficiency), it's still an important indicator to show the fragmentation of disk reading.

For W_1, the average chunk size is 14.8MB for Geo-1M, 25.0MB for Geo-4M, 56.4MB for Geo-16M, while the average chunk size is only 10.3MB for Stripe-Max layout. To recover, Stripe-Max needs 2.4x the number of disk seeks compared to Geo-4M. Thus *Geometric Partitioning* can utilize disk better than Stripe-Max during recovery. For W_2, the average chunk size is 324.8KB for Geo-128K, 622.4KB for Geo-256K.

Network and Disk Bandwidth for Recovery. We measure the average disk bandwidth and average network bandwidth for recovery in Table 3. Average disk bandwidth is the average amount of data read and write to a disk divided by repair time. Average network bandwidth is the average amount of network transfer to a node divided by repair time.

The gap between the measured and ideal disk bandwidth in Figure 4 is due to unbalanced (both spatial and temporal) and congested disk I/O, discontinuous reads, and extra software overhead in *RCStor*. We can see that network is far from full utilization, indicating that the network is not the bottleneck for recovery. *Geometric Partitioning* has higher disk bandwidth than Stripe layout, this is because *Geometric Partitioning* has larger chunk sizes. *Geometric Partitioning* recovers faster than RS code, Hitchhiker code, LRC code, but

Table 3. Disk and network bandwidth (in MB/s) by different schemes for W_1 and W_2 during recovery.

Scheme	Disk	Network	Disk	Network
	W_1		W_2	
Geo-1M	65	298	-	-
Geo-4M	79	369	-	-
Geo-16M	86	413	-	-
Geo-128K	-	-	532	451
Geo-256K	-	-	568	473
Con-16M	53	246	-	-
Con-64M	71	329	-	-
Con-256M	87	404	-	-
Con-128K	-	-	398	304
Con-512K	-	-	420	322
Stripe	25	115	120	92
Stripe-Max	57	263	264	202
RS	110	599	524	476
HH	89	466	493	427
LRC	95	487	549	467
ECPipe	113	616	534	485

uses less network bandwidth, implying that the recovery of *Geometric Partitioning* is more efficient.

Degraded Read Time by Sizes. We test the degraded read time for different workloads, layouts, and object sizes. We evaluate the 5*th*, median, and 95*th* time respectively. We show the impact of different layouts in Figure 11 and Figure 12. We can find that Geometric Partitioning provides not only good median degraded time but also good tail (95th) degraded read time, for different object sizes.

Pipelining by Client Bandwidth. We vary the bandwidth of the client to show how *Geometric Partitioning* benefits from pipelining under different network conditions. Figure 13 shows the average time to send repaired object to the client, the average time to repair an object, and the average time to degraded read an object when client bandwidth is 1Gbps, 2Gbps, and 4Gbps respectively. Figure 13 indicates that the degraded read time of *Geometric Partitioning* can be close to transfer time when network bandwidth is relatively low, close to repair time when network bandwidth is higher. The average degraded read time is reduced by 23.4%-35.9% by pipelining. For W_2, repair time dominates the total degraded read time (due to I/O latency, synchronization, software, etc.), so pipelining is less significant. However, *Geometric Partitioning* still helps by eliminating read amplification and enlarging chunk sizes.

Degraded Read Time for Range Access. We randomly specify a length and find a random corresponding offset to measure the average degraded read time of *Geometric*

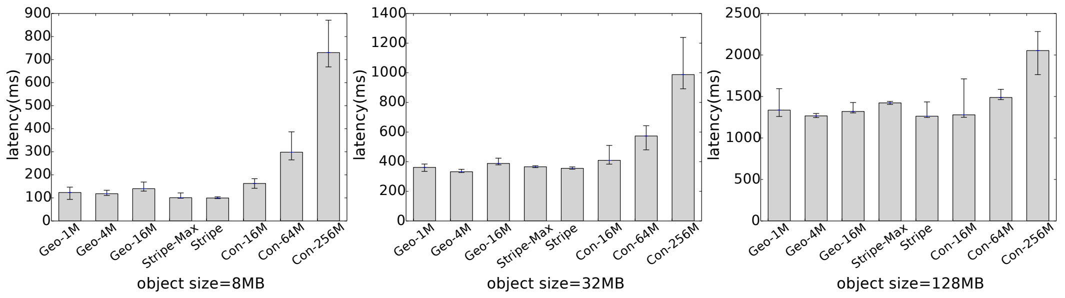


Figure 11. Degraded read time for different object sizes for W_1.

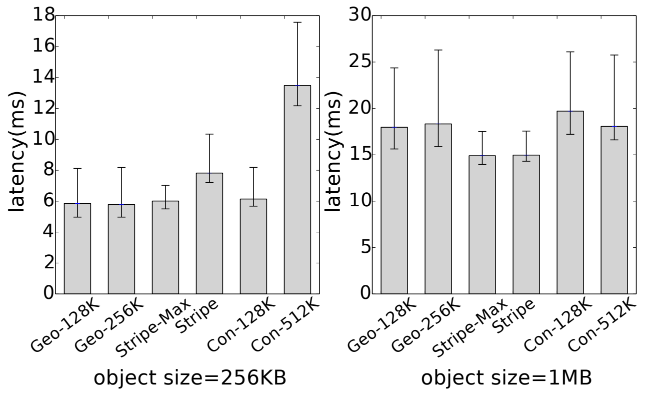


Figure 12. Degraded read time for different object sizes for W_2.

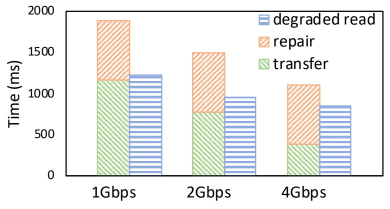


Figure 13. Average transfer time, repair time, and degraded read time of Geo-4M for different client bandwidth for W_1.

Table 4. Comparison of range degraded reads for different layouts.

	Read Size	Pipelining
Geometric	Less than object size	Sometimes
Contiguous	Possibly larger than object size	Sometimes
Stripe-Max	Equal to object size	No

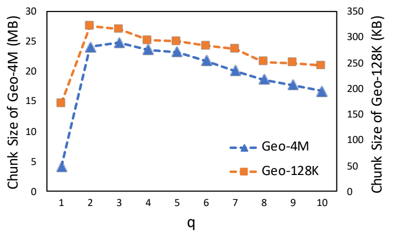


Figure 14. Average chunk size of Geo-4M for W_1, Geo-128K for W_2 under different q.

Partitioning for range access. The lengths we use to evaluate range accesses follow a uniform distribution, so the average length of a range access is half the size of the object. We measure the average degraded read time for W_1 when the system is idle, and we found that the degraded read time of Geo-4M is 67.6% of Con-16M, 55.3% of Stripe-Max. For W_2, the degraded read time of Geo-128K is 68.1% of Con-128K, 66.2% of Stripe-Max. As Table 4 shows, *Geometric Partitioning* only reads a portion of an object, while Stripe-Max needs to read the whole object and Contiguous need to read extra large chunks for degraded rage access. *Geometric Partitioning* also provides better pipelining than Stripe-Max layout.

Varying q. We vary q to show how q affects recovery. We can conclude from Figure 14 that the average chunk size peaks when $q = 2$ or $q = 3$. Thus setting q as 2 is a proper choice.

7 Related Work

Applying Regenerating Codes into Practice. There have been many attempts to apply regenerating codes into real storage systems [25, 27, 33, 38, 44]. However, they mainly focus on applying some specific regenerating codes into batch processing systems like HDFS [16], and lack the consideration for degraded read time and pipelining. Vajha et al. [44] attempt to implement Clay code in Ceph, where they found that disk I/O is saved when the system contains only 64MB objects. When there are objects of different sizes, disk I/O is not saved anymore due to read amplification.

Hybrid Strategies. Panasas [46] employs different RAID schemes for different objects. Giza [11] advocates using erasure codes only for objects larger than 4MB. EC-store [1], AutoRAID [47] hybrids erasure coding and replication to improve performance. These techniques focus on traditional

Table 5. Comparison of different layouts.

	Geometric	Stripe		Contiguous	
Chunk Size		Small	Large	Small	Large
Pipelining	Efficient	Efficient	Not efficient	Medium	Not efficient
Read amplification	No	No	No	Medium	Severe
Disk throughput for recovery	High	Low	Medium	Medium	High

RS codes and are not able to solve the challenges for regenerating codes.

Xia et.al. [48] proposes to use a code with faster recovery and lower storage efficiency for hot data and use a compact code with higher storage efficiency and slower recovery for cold data. In fact, MSR codes like Clay code are both optimal for storage efficiency and repair, so such kind of hybrid is not necessary anymore.

Geometric Sequences. Buddy system [23] exploits geometric sequences to support dynamic memory allocation. Dartmouth [24] uses a buddy allocator to manage disk space in its file system. *Geometric Partitioning* differs from buddy systems by introducing object partition. Buddy system partitions continuous regions to fit objects, while *Geometric Partitioning* partitions object to fit continuous regions.

Network Pipelining. *PPR* [28] and *ECPipe* [26] reduce repair time by eliminating network congestion on the data collection node. Instead of trying to reduce network bandwidth consumption, they reduce repairing time by distributing network traffic more evenly across the cluster. However, they are designed for erasure codes that satisfy addition associativity, which does not hold for Clay code.

8 Discussion

Geometric Partitioning needs to read more data than Stripe layout for degraded reads. However, they can be hidden by the transfer time through an efficient pipelining design, and as a result, the degraded read time of *Geometric Partitioning* can be much less than Stripe-Max. In addition, *Geometric Partitioning* puts each object into a single disk instead of k disks. As a result, if there are k degraded read requests for Stripe layout, there will be only 1 degraded read request for *Geometric Partitioning*. Thus *Geometric Partitioning* puts much less burden on CPU than Stripe layout, and its extra burden on disk and network for degraded read is compensated by much faster recovery because read requests will be degraded in a shorter period.

RCStor is designed to store immutable objects. It is possible to build key-value stores or databases by using object storage as the underlying storage engine to store tables and logs [4, 7, 14]. More effort is definitely needed to improve performance.

Though regenerating codes consume less network bandwidth than LRC codes and can recover at a higher efficiency, LRC codes can provide better locality. This is useful for cases like cross-data-center erasure coding because of reduced cross data center network traffic. Regenerating codes can still help by using LRC codes on top of regenerating codes [21], providing both better locality and less network bandwidth consumption. Our work is orthogonal to these works.

9 Conclusion

After revealing the practical gaps of applying regenerating codes in large-scale object storage systems, we propose geometric partitioning that split each object into chunks with sizes from a geometric sequence. It resolves the conflict between degraded read time and recovery efficiency. Table 5 summarizes the comparison of different layouts. We validate its advantages experimentally using real-world traces.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Matei Zaharia, for their valuable comments and helpful suggestions. Yingdi Shan, Kang Chen, Tuoyu Gong and Yongwei Wu are with the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2020YFC1522702), Natural Science Foundation of China (61877035).

References

[1] Michael Abebe, Khuzaima Daudjee, Brad Glasbergen, and Yuanfeng Tian. 2018. EC-Store: Bridging the Gap Between Storage and Latency in Distributed Erasure Coded Systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 255–266.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (Seattle, WA, USA) *(SIGCOMM '08)*. Association for Computing Machinery, New York, NY, USA, 63–74.

[3] Amazon. 2020. Amazon S3. https://aws.amazon.com/s3/.

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

[5] SB Balaji and P Vijay Kumar. 2018. A tight lower bound on the sub-packetization level of optimal-access MSR and MDS codes. In *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2381–2385.

[6] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. 2010. Finding a Needle in Haystack: Facebook's Photo Storage..

In *OSDI*, Vol. 10. 1–8.

[7] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* 251–264.

[8] Viveck R Cadambe, Cheng Huang, Syed A Jafar, and Jin Li. 2011. Optimal repair of MDS codes in distributed storage via subspace interference alignment. *arXiv preprint arXiv:1106.1250* (2011).

[9] Henry CH Chen, Yuchong Hu, Patrick PC Lee, and Yang Tang. 2014. NCCloud: A Network-Coding-Based Storage System in a Cloud-of-Clouds. *IEEE Trans. Computers* 63, 1 (2014), 31–44.

[10] Minghua Chen, Cheng Huang, and Jin Li. 2007. On the maximally recoverable property for multi-protection group codes. In *Information Theory, 2007.* IEEE, 486–490.

[11] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. 2017. Giza: Erasure Coding Objects across Global Data Centers. In *2017 USENIX Annual Technical Conference (USENIX ATC 17).* USENIX Association, 539–551.

[12] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. 2006. Efficient Replica Maintenance for Distributed Storage Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)* (San Jose, CA).

[13] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. 2004. Designing a DHT for Low Latency and High Throughput. In *First Symposium on Networked Systems Design and Implementation (NSDI 04).* USENIX Association, San Francisco, CA.

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

[15] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. 2010. Network coding for distributed storage systems. *IEEE Transactions on Information Theory* 56, 9 (2010), 4539–4551.

[16] Apache Software Foundation. 2018. Hadoop HDFS. http://hadoop.apache.org/.

[17] Eyal En Gad, Robert Mateescu, Filip Blagojevic, Cyril Guyot, and Zvonimir Bandic. 2013. Repair-optimal MDS array codes over GF (2). In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on.* IEEE, 887–891.

[18] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. 2012. On the locality of codeword symbols. *IEEE Transactions on Information Theory* 58, 11 (2012), 6925–6934.

[19] Sreechakra Goparaju, Arman Fazeli, and Alexander Vardy. 2017. Minimum Storage Regenerating Codes for All Parameters. *IEEE Transactions on Information Theory* 63, 10 (2017), 6318–6328. https://doi.org/10.1109/TIT.2017.2690662

[20] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. 2012. Erasure Coding in Windows Azure Storage. In *Usenix annual technical conference.* Boston, MA, 15–26.

[21] Govinda M Kamath, N Prakash, V Lalitha, and P Vijay Kumar. 2013. Codes with local regeneration. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on.* IEEE, 1606–1610.

[22] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *FAST.* 20.

[23] Kenneth C Knowlton. 1965. A fast storage allocator. *Commun. ACM* 8, 10 (1965), 623–624.

[24] Philip DL Koch. 1987. Disk file allocation based on the buddy system. *ACM Transactions on Computer Systems (TOCS)* 5, 4 (1987), 352–370.

[25] Katina Kralevska, Danilo Gligoroski, Rune E Jensen, and Harald Øverby. 2017. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data* 4, 4 (2017), 516–529.

[26] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. 2017. Repair Pipelining for Erasure-Coded Storage. In *2017 USENIX Annual Technical Conference (USENIX ATC 17).* USENIX Association, Santa Clara, CA, 567–579.

[27] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. 2019. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19).* USENIX Association, Boston, MA, 331–344.

[28] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. 2016. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems.* ACM, 30.

[29] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. F4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation.* USENIX Association, 383–398.

[30] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat datacenter storage. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12).* 1–15.

[31] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm.. In *USENIX Annual Technical Conference.* 305–319.

[32] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. 2013. The quantcast file system. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1092–1101.

[33] Lluis Pamies-Juarez, Filip Blagojevic, Robert Mateescu, Cyril Guyot, Eyal En Gad, and Zvonimir Bandic. 2016. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *FAST.* 81–94.

[34] Dimitris S Papailiopoulos, Jianqiang Luo, Alexandros G Dimakis, Cheng Huang, and Jin Li. 2012. Simple regenerating codes: Network coding for cloud storage. In *INFOCOM, 2012 Proceedings IEEE.* IEEE, 2801–2805.

[35] James S Plank, Kevin M Greenan, and Ethan L Miller. 2013. Screaming fast Galois field arithmetic using intel SIMD instructions.. In *FAST.* 299–306.

[36] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-cache: load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16).* USENIX Association, 401–417.

[37] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. 2015. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth.. In *FAST.* 81–94.

[38] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 331–342.

[39] Korlakai Vinayak Rashmi, Nihar B Shah, and P Vijay Kumar. 2011. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory* 57, 8 (2011), 5227–5239.

[40] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[41] Birenjith Sasidharan, Gaurav Kumar Agarwal, and P Vijay Kumar. 2015. A high-rate MSR code with polynomial sub-packetization level.

In *Information Theory (ISIT), 2015 IEEE International Symposium on*. IEEE, 2051–2055.

[42] Bianca Schroeder and Garth A Gibson. 2007. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?. In *FAST*, Vol. 7. 1–16.

[43] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. 2013. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory* 59, 3 (2013), 1597–1616.

[44] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. 2018. Clay codes: moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies*. 139.

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

[46] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *FAST*, Vol. 8. 1–17.

[47] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. 1996. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 108–136.

[48] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David Pease. 2015. A Tale of Two Erasure Codes in HDFS. In *FAST*. 213–226.