Clay Codes: Moulding MDS Codes to Yield an MSR Code

Abstract

With increasing scale, the number of node failures in a data center increases sharply. To ensure availability of data, failure tolerance schemes such as Reed-Solomon (RS) or more generally, *Maximum Distance Separable (MDS)* erasure codes are used. However, while RS or MDS codes offer minimum storage overhead for a given number of failure tolerance, they do not meet other practical needs of today's data centers. While modern codes such as minimum storage regenerating (MSR) codes are designed to meet these practical needs, these are available only in highly-constrained theoretical constructions, that are not sufficiently mature for practical implementation. We present *Clay codes* that extract the best from both worlds. Clay (short for Coupled-layer) codes are MSR codes that offer a simplified construction for decoding/repair by using pairwise coupling across multiple stacked layers of any single MDS code.

In addition, Clay codes provide the first practical implementation of an MSR code that offers (a) low storage overhead, (b) optimal repair bandwidth, (c) low sub-packetization level, (d) support for both single and multiple-node repairs, (e) uniform repair performance of data and parity nodes while permitting faster & more efficient repair.

While all MSR codes are vector codes, none of the distributed storage systems support vector codes. We have modified Ceph to support any vector code, and our contribution is on its way to Ceph's master codebase. We have implemented Clay codes, and integrated it as a plugin to Ceph. Six example Clay codes were evaluated on a cluster of Amazon EC2 instances whose parameters were carefully chosen to match known erasure-code deployments in practice. A particular example code, with storage overhead 1.25x, is shown to reduce repair network traffic by a factor of 2.9 in comparison with RS codes and similar reductions are obtained for both repair time and disk-read.

1 Introduction

As data centers scale, the number of failures in storage subsystems increase [9] [16] [28]. In order to ensure data availability and durability, failure-tolerant solutions such as replication and erasure codes are used. It is important for these solutions to be highly efficient so that they incur low cost in terms of their utilization of storage, computing and network resources. This additional cost is considered an overhead, as the redundancy introduced for failure tolerance does not aid the performance of the application utilizing the data.

Increasingly, data centers have started to adopt erasure codes in place of replication for failure tolerance. A class of erasure codes known as maximum distance seperable (MDS) codes offer the same level of failure tolerance as replication codes at a minimal storage overhead. For example, Facebook [18] report reduced storage overheads of 1.4x by using Reed-Solomon codes, a popular MDS code, as opposed to 3x in traditional triple replication [11]. The disadvantage of erasure codes is their high repair cost. That is, in case of replication, when a node or a storage subsystem fails, an exact copy of the lost data can be copied from surviving nodes. However, in case of erasure codes, dependent data that is more voluminous in comparison with the lost data, is copied from surviving nodes; and the lost data is then computed by a repair node. This results in increased repair bandwidth and repair time.

A second class of erasure codes termed as minimum storage regenerating (MSR) codes offer all the advantages of MDS codes, but require lesser repair bandwidth. Until recently, MSR codes lacked several key desirable properties that are important for practical systems. For example, they lacked constructions of common erasure code configurations [23], [19], they were computationally more complex [13], they demonstrated non-uniform repair characteristics for different types of node failures [12], they were able to repair from a limited (one or two)

number of failures [19], etc. The first theoretical construction that offered all of the desirable properties of an MSR code was presented by Ye & Barg [32].

This paper presents Clay codes (short for Coupled-layer codes) that extend the theoretical construction presented by Ye & Barg with practical considerations. Clay codes are constructed by placing any MDS code in multiple layers, and performing pair-wise coupling across layers. Such a construction offers efficient repair with reduced repair bandwidth, causing Clay codes to fall in the MSR arena.

We implement Clay codes and make it available as open-source under LGPL. We also integrate Clay codes as a plugin with Ceph, a distributed object storage system. Ceph supports scalar erasure codes such as Reed-Solomon codes. However, it does not support vector codes. We modified Ceph to support any vector code, and our contribution is on its way to Ceph's master/trunk codebase [2].

In erasure codes terminology, scalar codes require block-granular repair data, while vector codes can work at the sub-block granularity for repair. Ceph's equivalent to an erasure coded block is a chunk of object. That is, Ceph supports chunk-granular repair data, while our contribution extended it to sub-chunk granularity. To the best of our knowledge, our contribution, when accepted by the Ceph community, will cause Ceph to become the first-ever distributed storage system to support vector codes. Also, Clay codes will become the first-ever implementation of an MSR code that provides all desirable practical properties, and which is integrated to a distributed storage system.

Our contribution includes (a) the construction of Clay codes as explained in Section 3, (b) the modification we made to Ceph in order support any vector codes, explained in Section 4, and (c) the integration of Clay codes as a plugin to Ceph, explained in Section 4. We conducted experiments to compare the performance of Clay codes with Reed-Solomon codes available in Ceph and the results are presented in Section 5. Our experiments confirm reductions of 66% and 70% respectively in network traffic and disk-read during repair of a node-failure in a Ceph cluster employed with a particular Clay code. Significant reductions in network traffic and disk-read are observed while recovering from multiple failures as well.

2 Background and Preliminaries

Erasure Code Erasure codes are an alternative to replication for ensuring failure tolerance in data storage. In an $[n,k]$ erasure-coded system, data pertaining to an object is first divided into k data chunks and then encoded to obtain $m = n-k$ parity chunks. When we do not wish to distinguish between data or parity chunk, we will simply refer to the chunk as a *coded chunk*. The collection of n coded chunks obtained after encoding, are stored in n distinct nodes. Here, by *node*, we mean an independent failure domain such as a disk or a storage node of a distributed storage system (DSS). The storage efficiency of an erasure code is measured by the *storage overhead*, defined as the ratio of the number of coded chunks n to the number of data chunks k. Every erasure code has an underlying finite field over which computations are performed. For the sake of simplicity, we assume here that the field is of size 2^8 and hence each element of the finite field can be represented by a byte[1]. It is convenient to differentiate at this point, between *scalar* and *vector codes*.

Scalar Codes Let each data chunk be comprised of L bytes. In the case of a *scalar* code, one byte from each of the k data chunks is picked and the k bytes are linearly combined in m different ways, to obtain m parity bytes. The resultant set of $n = k+m$ bytes so obtained is called a *codeword*. This operation is repeated in parallel for all the L bytes in a data chunk to obtain L codewords. This operation will also result in the creation of m parity chunks, each composed of L bytes (see Fig. 1). As mentioned above, each coded chunk is stored on a different node.

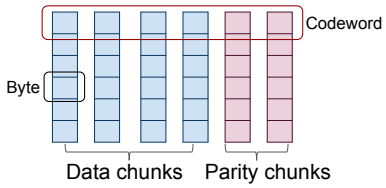


Figure 1: A pictorial representation of a scalar code. The $L = 6$ horizontal layers are the codewords and the $n = 6$ vertical columns, the chunks, with the first $k = 4$ chunks corresponding to data chunks and the last $(n-k) = 2$ chunks, the parity chunks. Each unit (tiny rectangle) in the figure corresponds to a single byte.

Vector Codes A similar process is used in the case of vector codes, with the difference that in the case of a *vector* code, one works with ordered collections of $\alpha \geq 1$ bytes at a time. For convenience, we will refer to such an ordered collection of α bytes as a *superbyte*. In the encoding process, a superbyte from each of the k data chunks is picked and the k superbytes are then linearly combined in m different ways, to obtain m parity superbytes. The resultant set of $n = k+m$ superbytes is called a (vector) *codeword*. This operation is repeated in parallel for all the $N = \frac{L}{\alpha}$ superbytes in a data chunk to obtain N codewords. Figure 2 shows a simple example where each superbyte consists of just two bytes.

[1]The codes described in this paper can however, be constructed over a finite field whose size is significantly smaller, and approximately equal to the parameter n. Apart from simplicity, we use the word byte here since the finite field of size 2^8 is a popular choice in practice.

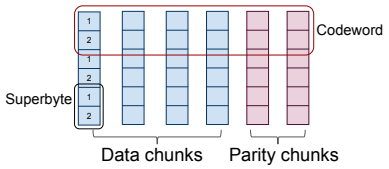


Figure 2: A pictorial representation of a vector code where each superbyte consists of 2 bytes. The picture shows $N = 3$ codewords. Each chunk, either data or parity, stores 3 superbytes, one corresponding to a different codeword.

The number α of bytes within a superbyte is called the level of sub-packetization of the code. Scalar codes such as Reed-Solomon codes can be regarded as having sub-packetization level $\alpha = 1$. Seen differently, one could also view a vector code as replacing α scalar codewords with a single vector codeword. The advantage of vector codes is that repair of the coded chunk in a failed node, can potentially be accomplished by accessing only a subset of the α bytes that are present in the remaining coded chunks that correspond to the same codeword. This leads to reduced network traffic during node repair.

Sub-chunking through Interleaving In Fig. 2, we have shown the α bytes associated to a superbyte as being stored contiguously. When the sub-packetization level α is large, given that operations involving multliple codewords are carried out in parallel, it is advantageous, from an ease-of-memory-access viewpoint, to interleave the bytes so that the corresponding bytes across different codewords are stored contiguously as shown in Fig. 3. This is particularly true, when the number N of superbytess within a chunk is large, for example, when $L = 8KB$ and $\alpha = 2$, contiguous access to $N = 4K$ bytes is possible. With interleaving, each data chunk is partitioned into α subsets, which we shall refer to as sub-chunks. Thus each sub-chunk within a node, holds one byte from each of the N codewords stored in the node.

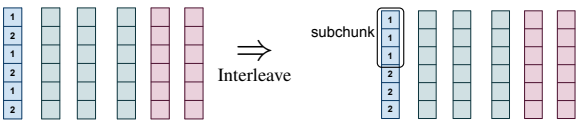


Figure 3: This figure shows the interleaving of the corresponding bytes within a superbyte across codewords, for the particularly simple case of two bytes within a superbyte. This results in a partitioning of the data chunk into sub-chunks and can lead to improved-memory-access performance.

MDS Codes The sub-class of (n,k) erasure codes, either scalar or vector, having the property that they can recover from the failure of any $(n-k)$ nodes are called Maximum Distance Separable (MDS) codes. These codes have the largest storage efficiency $\frac{n}{k}$ of any erasure code that can recover from the failure of a fixed number $(n-k)$ of node failures. Examples include Reed-Solomon (RS), Row-Diagonal Parity [7] and EVEN-ODD [5] codes. Other examples can be found described in [4]. Facebook data centers [27] for example, have employed an $[14,10]$ RS code in their data warehouse cluster.

Node Repair The need for node repair in a distributed storage system, can arise either because a particular hardware component has failed, is undergoing maintenance, is being rebooted or else, is simply busy serving other simultaneous requests for data. A substantial amount of network traffic is generated on account of node-repair operations. An example cited in [27], is one of a Facebook data-warehouse, that stores multiple petabytes of data, where the median amount of data transferred through top-of-rack switches for the purposes of node repair, is in excess of 0.2 petabytes per day. The network traffic arising from node-repair requests, eats into the bandwidth available to serve user requests for data. In addition, the time taken for node repair directly affects system availability. Thus there is strong interest in coding schemes that minimize the amount of data transfer across the network, and the time taken during the repair of a failed node. Under the conventional approach to repairing an RS code for instance, one would have to download k times the amount of data as is stored in a failed node to restore the failed node, which quite clearly, is inefficient.

MSR Codes MSR codes [8] are a sub-class of vector MDS codes that have the smallest possible repair bandwidth. To restore a data chunk in a failed node in an (n,k) MSR code, the code first contacts an arbitrarily-chosen subset of d helper nodes. The quantity d is a design parameter that can take on values ranging from k to $(n-1)$. It then downloads $\beta = \frac{\alpha}{d-k+1}$ bytes from each helper node in such a way that the total amount $d\beta$ of bytes downloaded is typically much smaller than the total amount $k\alpha$ bytes of data stored in the k data chunks. Here α is the sub-packetization level of an MSR code. The total number $d\beta = \frac{d\alpha}{(d-k+1)}$ of bytes downloaded for node repair, is called the *repair bandwidth*. Let us define the *normalized* repair bandwidth to be the quantity $\frac{d\beta}{k\alpha} = \frac{d}{k(d-k+1)}$. The normalization by $k\alpha$ can be motivated by viewing a single MSR codeword having sub-packetization level α as a replacement for α scalar RS codewords. The download bandwidth under the conventional repair of a α, scalar RS codes is precisely equal to $k\alpha$ bytes. For the particular case $d = (n-1)$, the normalized value equals $\frac{n-1}{k(n-k)}$. It follows that the larger the number $(n-k)$ of parity chunks, the greater the reduction in repair traffic. We will also use the parameter $M = k\alpha$ to denote the total number of databytes contained in an MSR codeword. Thus an MSR code has associated parameter set given by $\{(n,k),d,(\alpha,\beta),M\}$ with $\beta = \frac{\alpha}{d-k+1}$ and $M = k\alpha$.

Code	Storage O/h	Failure Tolerance	All node optimal repair	Access Optimal	Repair BW Optimal	α	Order of GF	Distributed Systems Implemented
RS	Low	$n-k$	No	No	No	1	Low	HDFS, Ceph, Swift, etc.
PM-RBT [23]	High	$n-k$	Yes	Yes	Yes	Linear	Low	Own system
Butterfly [19]	Low	2	Yes	No	Yes	Exponential	Low	HDFS, Ceph
HashTag [12]	Low	$n-k$	No	No	Yes	Polynomial	High	HDFS
Clay code	Low	$n-k$	Yes	Yes	Yes	Polynomial	Low	Ceph

Table 1: Detailed comparison of Clay codes with RS and other practical MSR codes. Here, the scaling of α is with respect to n for a fixed storage overhead (n/k).

Additional Desired Attributes: Over and above the low repair-bandwidth and low storage-overhead attributes of MSR codes, there are some additional properties that one would like a code to have. These include (a) uniform-repair capability, i.e., the ability to repair data and parity nodes with the same low repair bandwidth, (b) minimal disk read, meaning that the amount of data read from disk for node repair in a helper node is the same as the amount of data transferred over the network from the helper node and (c) low value of sub-packetization parameter α, and (d) a small size of underlying finite field over which the code is constructed.In MSR codes that possess the disk-read optimal property, both network traffic and number of disk reads during node repair are simultaneously minimized and are the same.

2.1 Related Work

The problem of efficient node repair has been studied for a while and several solutions have been proposed. Locally repairable codes such as the Windows Azure Code [14] and Xorbas [27] trade off the MDS property to allow efficient node-repair by accessing a lesser number of helper nodes. The piggy-backed RS codes introduced in [25] achieve reductions in network traffic while retaining the MDS property but they do not achieve the savings that are possible by an MSR code.

Though there were multiple implementaions of MSR codes, they were lacking in one or the other of the desired attributes (see Table 1). In [6], authors present FMSR codes for $(n-k)=2$, that allow efficient repair, but the data that is reconstructed during repair will not remain the same as the lost data. Consequently, it becomes necessary to invoke decoding algorithm at every read request. In [23], the authors implement a modified product-matrix MSR construction [26]. Although the code displays optimal disk IO performance, the storage overhead is on the higher side and of the form $(2-\frac{1}{k})$. In [19], the authors implement an MSR code known as the Butterfly code and experimentally validate the theoretically-proven benefits of reduced data download for node repair. However, the Butterfly code is limited to $(n-k)=m=2$ and has large value of sub-packetization 2^{k-1}. The restriction to small values of parameter m limits the efficiency of repair, as the normalized repair bandwidth can be no smaller than $\frac{1}{2}$. In this sense, the search for an MSR code having all of the desirable properties described above continued to remain elusive.

The recent theoretical results of Ye and Barg [32] have resulted in an altered situation. In this work, the authors provide a construction that permits storage overhead as close to 1 as desired, sub-packetization level close to the minimum possible, finite field size no larger than n, optimal disk IO, and all-node optimal repair. Clay codes offer a practical perspective and an implementation of the Ye-Barg's theoretical construction, along with several additional attributes. In other words, Clay codes possess all of the desirable properties mentioned above, and also offer several additional advantages compared to the Ye-Barg code.

2.2 Refinements over Ye-Barg Code

The presentation of the Clay code here is from a coupled-layer perspective that leads directly to implementation, whereas the description in [32] is primarily in terms of parity-check matrices. For example, using the coupled-layer viewpoint, both data decoding (by which we mean recovery from a maximum of $(n-k)$ erasures) as well as node-repair algorithms can be described in terms of two simple operations: (a) decoding of the scalar MDS code, and (b) an elementary linear transformation between pairs of bytes 3. While this coupled-layer viewpoint was implicit in the Ye-Barg paper [32], we make it explicit here.

In addition, Clay codes can be constructed using any scalar MDS code as building blocks, while Ye-Barg code is based only on Vandermonde-RS codes. Therefore, scalar MDS codes that have been time-tested, and best suited for a given application or workload need not be modified in order to make the switch to MSR codes. By using Clay codes, these applications can use the same MDS code in a coupled-layer architecture and get the added benefits of MSR codes.

The third important distinction is that, in [32], only one node failure cases are discussed. For Clay codes, we studied repair cases for multiple node failures as well.

Also, we came up with a generic algorithm to repair multiple failures.

3 Construction of the Clay Code

We will introduce the Clay code through an illustrative example. While in Section 2, we noted that each node stores a data chunk and that a data chunk is comprised of N codewords, our description in the present section will restrict to the case of a single codeword, i.e., to the case when $N = 1$. As explained in Section 2, by interleaving across the N codewords, one can make the transition from bytes to sub-chunks and it is at the granularity of a sub-chunks that operations take place in practice.

Our example code will have parameters: $\{(n = 6, k = 4),\ d = 5, (\alpha = 8, \beta = 4),\ M = 32\}$

(n,k)	d	(α,β)	$(d\beta)/(k\alpha)$
(6,4)	5	(8,4)	0.625
(12,9)	11	(81,27)	0.407
(14,10)	13	(256,64)	0.325
(14,10)	11	(128,64)	0.55
(20,16)	19	(1024,256)	0.297

Table 2: A list of example parameters of a Clay code along with the corresponding normalized repair bandwidth. Each of the codes in this table are experimentally evaluated in the present work.

The codeword is stored across $n = 6$ nodes. Each superbyte in this vector code, is comprised of $\alpha = 8$ bytes, with one data superbyte stored in each of the $k = 4$ data nodes, while the $(n-k) = 2$ parity nodes each store one parity superbyte. The storage efficiency of the code is the ratio $\frac{n\alpha}{k\alpha} = \frac{n}{k} = 1.5$ of the total number $n\alpha = 48$ of bytes stored to the number $M = k\alpha = 32$ of data bytes. During repair of a failed node, $\beta = 4$ bytes of data is downloaded from each of the $d = 5$ helper nodes needed to recover the lost superbyte. The normalized repair bandwidth is thus given by $\frac{(n-1)}{k(d-k+1)} = 0.625$.

Table 2 lists the parameters of some other Clay codes that can be constructed and as can be seen, the normalized repair bandwidth can be made much smaller, by increasing the value of $(d-k+1)$. For example the normalized repair bandwidth for a $(20,16)$ code equals 0.297, meaning that the amount of data download for node repair in the case of a Clay code, is less than 30% of the corresponding value for α layers of an RS code.

Data Cube Representation It will be convenient to use a two-dimensional representation, in terms of (x,y) coordinates to index the 6 nodes,

$$\{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)\},$$

as opposed to a linear numbering $\{1,2,\cdots,6\}$. Thus the 6 nodes can be visualized as forming a (2×3) grid (Fig. 4). We will say that nodes belong to the same *y-section* if they share the same y coordinate. We picture

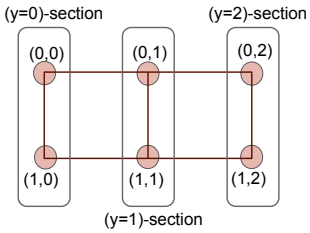


Figure 4: The (x,y) representation of nodes showing y-sections.

these $\alpha = 8$ bytes forming the superbyte in each node as extending along a third z dimension. This leads us to a 3D pictorial representation for the data. We will refer to this as the *data cube* associated with the codeword being stored (Fig. 5). With this, we can think of the data cube as comprised of 8 horizontal layers each indexed by the integer $z \in \{0,1,\cdots,7\}$. It will be found convenient at times to replace the scalar index z, by its corresponding binary representation (z_0,z_1,z_2) as below:

$$\{0 \Rightarrow (0,0,0),\ 1 \Rightarrow (0,0,1),\ \cdots,\ 6 \Rightarrow (1,1,0),\ 7 \Rightarrow (1,1,1)\}.$$

Thus we can also associate each horizontal layer with the binary 3-tuple (z_0,z_1,z_2). We use a pattern of black dots to indicate the label associated with each layer as shown in Fig 5 which shows the layer $(z_0,z_1,z_2) = (0,1,0)$.

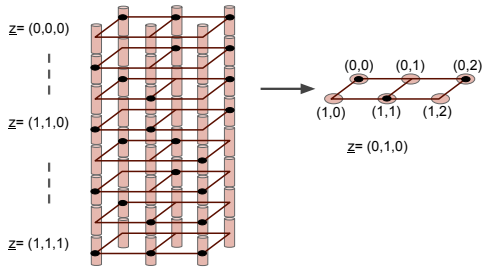


Figure 5: Each vertical column in the data cube corresponds to a node and each node stores a superbyte comprising of 8 bytes. Each of the smaller sub-cylinders into which the contents of a node broken up, correspond to a single byte. The figure on the right illustrates the use of a pattern of black dots to identify a horizontal plane, or equivalently, a byte within a superbyte.

We have thus far, indicated that the data stored in the nodes and associated to a single codeword is best viewed as corresponding to a data cube with one byte stored on each vertex of the data cube. It remains to explain how redundancy is introduced into this structure and how encoding, decoding and node repair are carried out.

Pairing of Bytes within the Data Cube Each vertex $(x,y;z)$ of the datacube is associated to a byte $A(x,y;z)$ of data. The 8 bytes $\{A(x,y;z)\}$ sharing the same (x,y)

coordinate represent the superbyte stored in node (x,y). There are two classes of vertices, those associated to a black dot (colored) and those that are not (uncolored), eqaully distributed at 24 apiece. The 24 uncolored vertices are paired as shown in Fig. 6. The pair is composed of a byte $A(x,y;z)$ and its companion $A^*(x,y;z)$. The coordinates of the companion of the byte associated to $(x,y;z)$ can be computed simply by representing z in binary form and interchanging the values of x and z_y. A few such pairings are shown in Table 3. Note that a symbol A and its companion A^* both belong to the same y-section.

Coordinates of vertex	Coordinates of companion vertex
(0, 0, 101)	(1, 0, 001)
(1, 1, 101)	(0, 1, 111)
(0, 2, 101)	(1, 2, 100)

Table 3: Three example pairings of vertices.

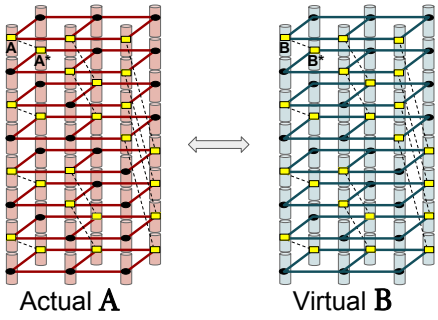


Figure 6: The virtual data cube can be obtained from the Actual data cube by a pairwise forward transformation and in the reverse direction, by a pairwise reverse transformation. Vertex pairs in both Virtual and Actual data cubes are identified by yellow rectangles connected using dotted lines.

Virtual Data Cube To explain the encoding process, we introduce a second data cube $\mathbb{B}$ of identical size and structure. To explain the relationship between the data cubes $\mathbb{A}$ and $\mathbb{B}$, we introduce a pairwise forward transformation (PFT) that maps bytes in the data cube $\mathbb{A}$ to corresponding bytes in the data cube $\mathbb{B}$ (see Fig. 6). We will refer to $\mathbb{A}$, $\mathbb{B}$ as the actual and virtual data cubes respectively. Given a byte A having a companion A^*, we define corresponding subchunks B, B^* in the $\mathbb{B}$ data cube via the linear transformation:

$$\begin{bmatrix} B \\ B^* \end{bmatrix} = \begin{bmatrix} 1 & u \\ u & 1 \end{bmatrix} \begin{bmatrix} A \\ A^* \end{bmatrix} \qquad (1)$$

where u is an element in the finite field of size 2^8 satisfying $u^2 \neq 1$ and $u \neq 0$. Here we have abbreviated and written A in place of $A(x,y;z)$ and similarly with $\{A^*, B, B^*\}$. Under this choice of u, it can be verified that, any two bytes in the set $\{A, A^*, B, B^*\}$ can be obtained from the remaining two. We will refer to the computation in the reverse direction as the pairwise reverse transformation(PRT). In the case of an unpaired byte A the relation is simply given by $A = B$.

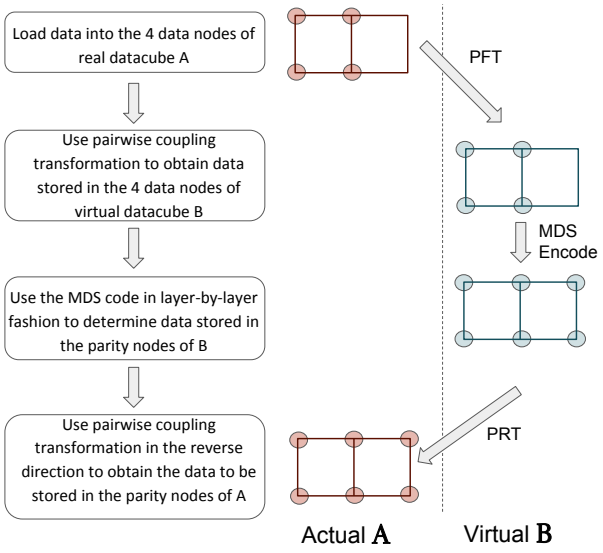


Figure 7: Encoding Flowchart for the Clay code. A topview of the nodes is shown on the right. The nodes colored in pink correspond to the real datacube and those in blue to the virtual datacube.

Constraints and Encoding The constraints placed on the data cube $\mathbb{A}$ arise indirectly from the pairwise relation ship between data cubes $\mathbb{A}$ and $\mathbb{B}$ and in addition, the key requirement that the symbols $\{B(x,y;z) \mid z \in \{0,1,\ldots,7\}$ in each horizontal layer of the data cube $\mathbb{B}$ are required to form an $(6,4)$ MDS code. While the layers in the data cube $\mathbb{B}$ are uncoupled, in the case of the data cube $\mathbb{A}$, the constraints on the data cube $\mathbb{B}$ in conjunction with the PFT causes coupling among the layers in $\mathbb{A}$ hence the name Coupled-Layer (Clay) code. The flow chart in Fig.7 provides a self-explanatory description of the encoding process. To explain node repair and decoding, we introduce the notion of an intersection score.

Intersection Score The Intersection Score (IS) of a layer is given by the number of hole-dot pairs in a plane, where holes correspond to erasures and by dots, we mean the back dots introduced earlier. For example in Fig. 8, when nodes $(0,1)$, $(0,2)$ are erased, layers $(0,0,0),(0,1,0),(0,1,1)$ have respective IS=$2,1,0$.

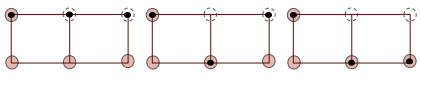


Figure 8: Illustration of the intersection score (IS) for erasures at $(0,1),(0,2)$.

Node Repair During node repair, all the layers with IS > 0 are picked and the bytes from these layers corresponding to all the remaining nodes represent the helper data. In Fig. 9 the repair of node $(1,0)$ is illustrated and

the corresponding repair layers are indicated. It can be seen that only 4 bytes from each of the 5 nodes participate in the repair process. Thus a total of 20 bytes suffice for node repair, whereas the repair of an RS code would require the transfer of all the data belonging to any 4 nodes, for a total data download of 32 bytes.

Decoding The decoding algorithm "Decode" of the Clay code is able to correct for the erasure of any any $m = n - k = 2$ nodes. Decoding is carried out sequentially, layer-by-layer, in order of increasing IS. We describe below as an example, the decoding algorithm when nodes $(0,1)$, $(0,2)$ are erased. In Fig.10, we explain the decoding procedure for a layer having IS$= 0$. A similar procedure can be followed subsequently for layers with IS $= 1$. followed by layers with IS$= 2$. The only difference is that in decoding a layer with IS> 0, we will need to use the fact that bytes in layers with lesser IS have already been decoded. The sequential decoding algorithm ensures that all the bytes B in the corresponding layer of $\mathbb{B}$ can be decoded. Having obtained all the B bytes, the A bytes can be computed using the pairwise reverse transformation (PRT).

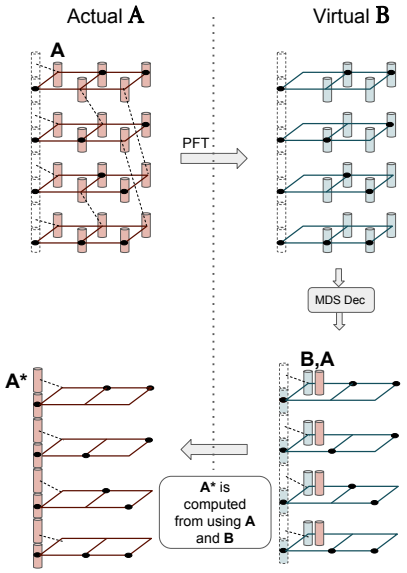


Figure 9: The dotted cylinder identifies the erased node. All the bytes shown on the top left represent helper data, the layers shown are the repair layers. The pairwise forward transform is first performed on the helper data to obtain the corresponding B bytes followed by the operation 'MDS decode to obtain the remaining B subchunks. The erased chunk is now computed using the PRT.

Clay code parameters Clay codes can be constructed in for any t and any parameter set of the form:

$$(n = qt,\ k,\ d)\ \ (\alpha = q^t, \beta = q^{t-1}),\ \text{with}\ q = (d-k+1).$$

The encoding, decoding and repair algorithms all are along the lines of that for the case $d = (n-1)$. However,

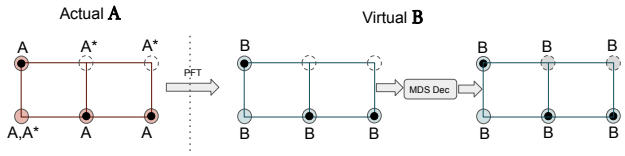


Figure 10: Illustrating how the Clay code recovers from the maximum possible 2 of erasures. (Left) the pink circles indicate the non-erased nodes. (Middle) blue nodes indicate nodes in the virtual data cube $\mathbb{B}$ whose contents can be determined through pairwise forward transformation(PRT), (Right) Through decoding of the MDS code, contents of all B nodes in this layer can be determined.

in the case $d < n-1$, during single node repair, while picking the d helper nodes, one must include among the d helper nodes, all the nodes belonging to the failed node's y-section.

Clay codes for any (n,k,d) The parameters indicated above have the restriction that q divide n. But the construction can be extended in a simple way to the case when q is not a factor of n. Our implementation of Clay code includes this generalization.

4 Ceph and Vector MDS Codes

4.1 Introduction to Ceph

Ceph [29] is a popular, open-source distributed storage system [30], that permits the storage of data as objects. Object Storage Daemon (OSD) is the daemon process of Ceph, associated with a storage unit such as a solid-state or hard-disk drive, on which user data is stored.

Ceph supports multiple erasure-codes, and a code can be chosen by setting attributes of the erasure-code-profile. Objects will then be stored in logical partitions referred to as pools associated with an erasure-code-profile. Each pool can have a single or multiple placement groups (PG) associated with it. A PG is a collection of n OSDs, where n is the block length of the erasure code associated to the pool.

The allocation of OSDs to a PG is dynamic, and is carried out by the CRUSH algorithm [31]. When an object is streamed to Ceph, the CRUSH algorithm allocates a PG to it. It also performs load balancing dynamically whenever new objects are added, or when active OSDs fail. Each PG contains a single, distinct OSD designated as the primary OSD (p-OSD). When it is required to store an object in a Ceph cluster, the object is passed on to the p-OSD of the allocated PG. The p-OSD is also responsible for initiating the encoding and recovery operations.

In Ceph, the passage from data object to data chunks by the p-OSD is carried out in two steps as opposed to the single-step description in Section 2. For a large object, the amount of buffer memory required to perform encoding and decoding operations will be high. Hence, as an

intermediate step, an object is first divided into smaller units called *stripes*, whose size is denoted by S (in bytes). If an object's size is not divisible by S, zeroes are padded. The object is then encoded by the p-OSD one stripe at a time. The stripe-size is to be specified within the cluster's configuration file. Both zero padding and system performance are important factors to be considered while fixing a stripe-size.

4.2 Sub-Chunking through Interleaving

To encode, the p-OSD first zero pads each stripe as necessary in order to ensure that the strip size S is divisible by $k\alpha$. The reason for the divisibility by a factor of k is because as described earlier, the first step in encoding is to break up each stripe into k data chunks of equal size. The reason for the additional divisibility requirement by a further factor α arises because we are dealing with a vector code and as explained in Section 2, operations in a vector code involve superbytes, where each superbyte contains α bytes. In what follows, we will assume that S is divisible by $k\alpha$.

The encoding of a stripe is thus equivalent to encoding $N = \frac{S}{k\alpha}$ codewords at a time. The next step as explained in Section 2, is interleaving at the end of which one obtains α sub-chunks per OSD, each of size N bytes. We note that the parameter L introduced in Section 2, is the number of bytes per data chunk and is thus given by $L = \frac{S}{k}$. This notion of sub-chunk is not native to Ceph, but rather is a modification to the Ceph architecture proposed here, to enable the support of vector codes.

The advantage of a vector code is that it potentially enables the repair of an erased coded chunk by passing on a subset of the α sub-chunks. For example, in the Clay code implemented in Ceph here is an MSR code, it suffices for each node to pass on β sub-chunks. However, when these β sub-chunks are not sequentially located within the storage unit, it can result in fragmented reads. We analyze such disk read performance degradation in Section 5.

4.3 Implementation in Ceph

Our implementation makes use of the Jerasure [21] and GF-Complete [20] libraries which provide implementations of various MDS codes and Galois-field arithmetic. We chose in our implementation to employ the finite field of size 2^8 to exploit the computational efficiency for this field size provided by the GF-complete library in Ceph.

In the our implementation, we employ an additional buffer, termed as *B-buffer*, that stores the sub-chunks associated with the virtual data cube $\mathbb{B}$ introduced in Section 3. This buffer is of size $n\alpha = S\frac{n}{k}$ bytes. The *B-buffer* is allocated once for a PG, and is used repetitively during encode, decode and repair operations of any object belonging to that PG.

Pairwise Transforms We introduced functions that compute any two sub-chunks in the set $\{A, A^*, B, B^*\}$ given the remaining two sub-chunks. We implemented these functions using the function *jerasure_matrix_dotprod()*, which is built on top of function *galois_w08_region_multiply()*.

Encoding Systematic encoding of an object is carried out by p-OSD by pretending that m parity chunks have been erased, and then recovering the m chunks using the k data chunks by initiating the decoding algorithm for the code. Pairwise forward and reverse transforms are the only additional computations required for Clay encoding in comparison with MDS encoding.

Enabling Selection Between Repair & Decoding When one or more OSDs go down, multiple PGs are affected. Within an affected PG, recovery operations are triggered for all associated objects. We introduced a boolean function *is_repair()* in order to choose between a bandwidth, disk I/O efficient repair algorithm and the default decode algorithm. For the case of single OSD failure, *is_repair()* always returns *true*. There are multiple failure cases as well for which *is_repair()* returns *true* i.e., efficient repair is possible. We discuss these cases in detail in Section.6.

Helper-Chunk Identification In the current Ceph architecture, when a failure happens, *minimum_to_decode()* is called in order to determine the k helper chunk indices. We introduced a function *minimum_to_repair()* to determine the d helper chunk indices when repair can be performed efficiently i.e., when *is_repair()* returns *true*. OSDs corresponding to these indices are contacted to get information needed for repair/decode. When there is a single failure, *minimum_to_repair()* returns d chunk indices such that all the chunks that fall in the y-cross-section of the failed chunk are included. We describe the case of multiple erasure cases in detail in Section.6

Fractional Read For the case of efficient repair, we only read a fraction of chunk, this functionality is implemented by feeding repair parameters to an existing structure *ECSubRead* that is used in inter-OSD communication. We have also introduced a new read function with Filestore of Ceph that supports sub-chunk reads.

Decode and Repair Either the decode or repair function is called depending on whether if *is_repair()* returns *true* or *false* respectively. The decoding algorithm is described in Sec. 3. Our repair algorithm supports in addition to single-node failure (Sec.3), some multiple-erasure failure patterns as well (Sec. 6).

4.4 Potential Contributions to Ceph

Enabling vector codes in Ceph : We introduced the notion of sub-chunking and the functions

get_repair_subchunks, *minimum_to_repair* in order to enable new vector erasure code plugins. This contribution is currently under review.

Clay codes in Ceph : We implemented Clay codes as a technique (*cl_msr*) within the *jerasure* plugin. The current implementation gives flexibility for a client to pick any n, k, d parameters for the code. It also gives an option to choose the MDS code used within to be either a Vandermonde-based-RS or Cauchy-original code.

5 Experiments and Results

The experiments conducted to evaluate the performance of Clay codes in Ceph while recovering from a single node failure are discussed in the present section. Experimental results relating multiple node-failure case can be found in Sec. 6.1.

5.1 Overview and Setup

Codes Evaluated While Clay codes can be constructed for any parameter set (n, k, d), we have carried out experimental evaluation for selected parameter sets close to those of codes employed in practice, see Table 4. Code C1 has (n, k) parameters comparable to that of the RDP code [7], Code C2 with the locally repairable code used in Windows Azure [15], and Code C3 with the $[20, 17]$-RS code used in Backblaze [1]. There are three other codes C4, C5 and C6 that are an approximate match for the $[14, 10]$-RS code used in Facebook data-analytic clusters [24]. Results relating to Codes C4-C6 can be found in Sec. 6.1, which focuses on repair in the multiple-erasure case.

	(n, k, d)	α	Storage overhead	$\frac{\beta}{\alpha}$
C1	(6,4,5)	8	1.5	0.5
C2	(12,9,11)	81	1.33	0.33
C3	(20,16,19)	1024	1.25	0.25
C4	(14,10,11)	128	1.4	0.5
C5	(14,10,12)	243	1.4	0.33
C6	(14,10,13)	256	1.4	0.25

Table 4: Codes C1-C3 are evaluated in Ceph for single-node repair. The evaluation of Codes C4-C6 is carried out for both single and multiple-node failures.

The experimental results for Clay code are compared against experimental results for RS codes possessing the same (n, k) parameters. By an RS code, we mean an MDS-code implementation based on the *cauchy_orig* technique of Ceph's *jerasure* plugin. The same MDS code is also employed as the MDS code appearing in the Clay-code construction evaluated here.

Experimental Setup All evaluations are carried out on Amazon EC2 instances of the m4.xlarge (16GB RAM, 4 CPU cores) configuration. Each instance is attached to an SSD-type volume of size 500GB. We integrated the Clay code in Ceph Jewel 10.2.2 to perform evaluations. The Ceph storage cluster deployed consists of 26 nodes. One server is dedicated for the MON daemon, while the remaining 25 nodes each run one OSD. Apart from the installed operating system, the entire 500GB disk is dedicated to the OSD. Thus the total storage capacity of the cluster is approximately 12.2TB.

Model	Object Distribution		Total, T (GB)	Stripe size, S
	Object size (MB)	# Objects		
Fixed (W_1)	64MB	8192	512	64MB
Variable (W_2)	64	6758	448	1MB
	32	820		
	1	614		

Table 5: Workload models used in experiments.

Overview Experiments are carried out on both fixed and variable object-size workloads, respectively referred to as W_1 and W_2. In the W_2 workload, we choose objects of sizes 64MB, 32MB and 1MB distributed in respective proportions of 82.5%, 10% and 7.5%. Our choices of object sizes cover a good range of medium (1MB), medium/large(32MB) and large (64MB) objects[3], and the distribution is chosen in accordance with that in the Facebook data analytic cluster reported in [22]. The workloads used for evaluation are summarized in Tab. 5. The stripe-size S is set as 64MB, and 1MB respectively, for fixed and variable object-size workloads so as to avoid zero-padding.

The failure domain is chosen to be a node. Since we have one OSD per node, this is equivalent to having a single OSD as the failure domain. We inject node failures into the system by removing OSDs from the cluster. Measurements are taken using *nmon* and *NMONVisualizer* tools. We run experiments with a single PG, and validate the results against the theoretical prediction. We also run the same experiments with 512 PGs, which we will refer to as the multiple-PG case. Measurements are made of (a) repair network traffic, (b) repair disk read, (c) repair time, and (d) encoding time.

5.2 Evaluations

Network Traffic: Single Node Failure Network traffic refers to the data transferred across the network during single-node repair. Repair is carried out by the p-OSD, which also acts as a helper node. The theoretical estimate for the amount of network traffic is $\frac{T}{k}((d-1)\frac{\beta}{\alpha}+1)$ bytes for a Clay code, versus T bytes for an RS code. Our evaluations confirm the expected savings, and we observed reductions of 25%, 52% and 66%, (a factor of 2.9×) in network traffic for codes C1, C2 and C3 respectively in comparison with the corresponding RS codes under fixed and variable workloads

(see Fig. 11(a), 11(d).) As can be seen, the code C3 with the largest value of $q = (d-k+1)$ offer the largest savings in network traffic.

In certain situations, an OSD that is already part of the PG can get reassigned as a replacement for the failed OSD, leading to inferior network-traffic performance in a multiple-PG setting. Nevertheless, we present the performance for a single run of the experiment with multiple PGs under the W_1 workload in Fig. 12. We emphasize however, that single-run performance cannot be used to make judgments on performance.

Disk Read: Single Node Failure The amount of data read from the disks of the helper nodes during the repair of a failed node is referred to as disk read and is an important parameter to minimize [23], [17].

Depending on the index of the failed node, the sub-chunks to be fetched from helper nodes in a Clay code can be contiguous or non-contiguous. Non-contiguous reads in HDD volumes lead to a slow-down in performance [19]. Even for SSD volumes that permit reads at a granularity of 4kB, the amount of disk-read needed depends on the sub-chunk-size. We explain here, disk reads from a helper node in the case of single node failure for the code C3 in case of workload W2 (S=1MB). The chunk size here is given by $L = S/k = 64$kB. During repair of any node, $L/(d-k+1) = 16$kB of data is to be read from each helper node. In the best case scenario, the 16kB data is contiguous, where as for the worst case scenario the reads are fragmented. In the later case, $\beta = 256$ fragments with each of size $L/\alpha = 64$bytes are read. When 4kB of data is read from device, only 1kB ends up being useful for the repair operation. Therefore, the data read essentially is 4 times the amount of information needed for repair. This is evident in disk reads from a helper node for the worst case in 11(f).

When the same code C3 is used for W1 workload (64MB stripe size), for the worst case scenario, fragments of size $S/k\alpha = 4$kB are read. As this size is aligned to the granularity of SSD reads, disk reads for worst case is equal to $256 * 4$kB=1MB. This is exactly the amount read during best case(see 11(f)). The experimental results in 11(e) suggest that for W2 workloads, C1 is a good choice, while C2 and C3 are not suitable. Whereas results in 11(b) show that all the three codes perform well for the case of W1 workloads.

While making disk-read measurements, we clear the cache prior to repair to obtain the actual disk-read. The expected disk-read from all helper nodes during repair is $\frac{Td\beta}{k\alpha}$ bytes for a Clay code in contrast to T bytes for an RS code. In experiments with fixed object-size (see Fig. 11(b)), we obtain savings of 37.5%, 59.3% and 70.2% (a factor of 3.4×) for codes C1, C2 and C3 respectively, when compared against the corresponding RS code. Fig. 12 shows the disk read in the multiple-PG setting.

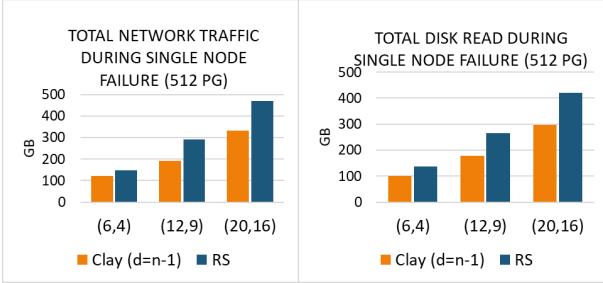


Figure 12: Network traffic and disk read during repair of single node in a setting with 512 PGs, for W_1 workload.

Repair Time and Encoding Time We measure the time taken for repair by capturing the starting and stopping times of network activity within the cluster. We observed a significant reduction in repair time for Clay codes in comparison with an RS code. For the code C3 in a single-PG setting, we observe a reduction by a factor of 3× in comparison with an RS code. This is mainly due to reduction in network traffic and disk IO required during repair (see Fig.11(c)).

We define the time required by the RADOS utility to place an object into Ceph object-store as the *encoding time*. The encoding time includes times taken for computation, disk-IO operations, and data transfer across the network. We define the time taken for computing the code chunks based on the encoding algorithm as the *encode computation time*. During encoding, the network traffic and I/O operations are the same for both the classes of codes. Although the encode computation time of Clay code is higher than that of the RS code (See Fig. 13.) the encoding time of a Clay code remains close to that of the corresponding RS code. The increase in the computation time for the Clay code is due to the multiplications involved in PFT and PRT operations. In storage systems, while data-write is primarily a one-time operation, failure is a norm and thus recovery from failures is a routine activity [10],[23]. The significant savings in network traffic and disk reads during node repair are a sufficient incentive for putting up with overheads in the encode computation time.

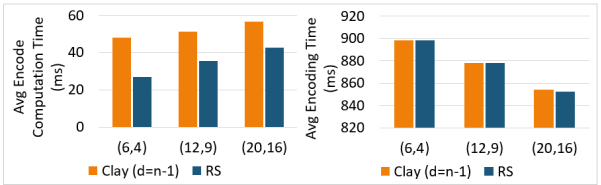
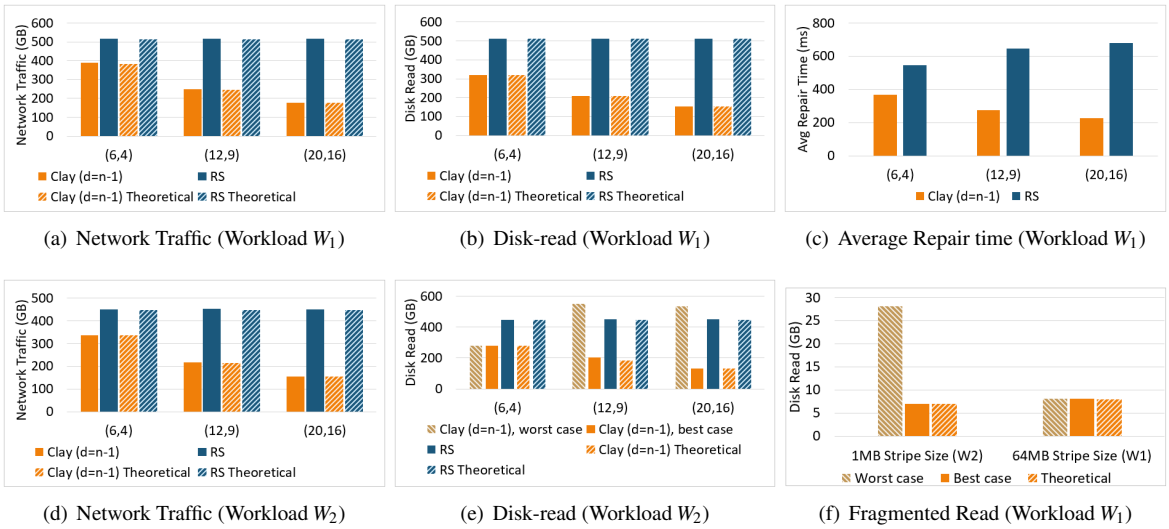


Figure 13: Comparison of average encoding times for C1, C2 and C3 in comparison with RS codes, for the W_1 workload.



(a) Network Traffic (Workload W_1) (b) Disk-read (Workload W_1) (c) Average Repair time (Workload W_1)

(d) Network Traffic (Workload W_2) (e) Disk-read (Workload W_2) (f) Fragmented Read (Workload W_1)

Figure 11: Experimental evaluation of C1, C2 and C3 in comparison with RS codes in a single-PG setting is presented in plots (a)-(e). The plot (f) gives a relative comparison of disk-read in a helper node for stripe-sizes 4MB and 64MB for code C3.

6 Handling Failure of Multiple Nodes

The Clay code is capable of recovering from multiple node-failures with savings in repair bandwidth. The failure patterns that can be recovered with bandwidth-savings are referred to as *repairable failure patterns*.

Repairable Failure Patterns (i) $d < n-1$: Clay codes designed with $d < n-1$ can recover from e failures with savings in repair bandwidth when $e \leq n-d$, with a minor exception described in Remark 1. We have to choose helper nodes in such a way that if a y-section contains a failed node, then all the surviving nodes in that y-section must act as helper nodes. For example, consider the code with parameters $(n=14, k=10, d=11)$. The nodes can be put in a (2×7) grid, as $q=d-k+1=2$ and $t=\frac{n}{q}=7$. In Fig.14, we assume that nodes $(0,0)$ and $(0,1)$ have failed, and therefore nodes $(1,0)$ and $(1,1)$ along with any 9 other nodes can be picked as helper nodes.

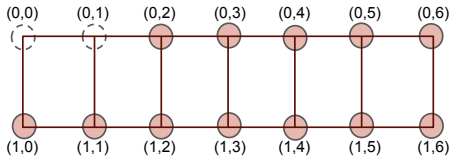


Figure 14: The (2×7) grid of 14 nodes in Clay code with parameters $(14,10,11)$. The nodes $(0,0)$ and $(0,1)$ have failed.

(ii) $d = n-1$: When the code is designed for $d=(n-1)$, any number of upto $(q-1)$ failures that occur within a single y-section can be recovered with savings in repair bandwidth. As the number of surviving nodes is smaller than d in such a case, all the surviving nodes are picked as helper nodes. See Fig. 15 for an example of failure-pattern in the case of a $(14,10,13)$ Clay code.

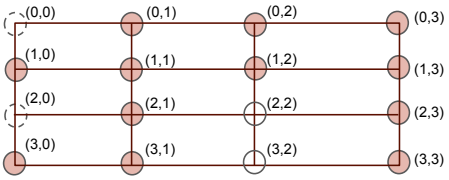


Figure 15: The (4×4) grid containing 14 nodes in $(14,10,13)$ Clay code. The cells $(2,2)$ and $(3,2)$ in the grid do not represent nodes. The nodes $(0,0)$ and $(2,0)$ coming from 0-section have failed.

Repair Layers For the case of single failure, we have already observed that all the layers with IS >0 are picked. The logic remains the same for case of multiple failures as well.

Remark 1 *Whenever there are q failures within the same y-section, every layer will have IS >0, and the repair algorithm will need all the sub-chunks from every helper node. Therefore in these cases, decode algorithm is a better option and the is_repair() function (see Sec. 4.3.) takes care of this case by returning false.*

Repair Bandwidth Savings We describe here how to compute network traffic during the repair of a multiple-failure pattern. Let e_i be the number of erased nodes within $(y=i)$-section, and the total number e of failures is given by $e=\sum_{i=0}^{t-1} e_i$. The number of helper nodes $d_m = d$ if the code is designed for $d<(n-1)$, and $d_m = n-e$ if it is designed for $d=(n-1)$. Total number of sub-chunks

β_m needed from each helper node is same as the number of layers with IS > 0. This can be obtained by subtracting from α the count of layers with IS$= 0$. The number of helper sub-chunks per node, $\beta_m = \alpha - \prod_{i=0}^{t-1}(q - e_i)$, and network traffic for repair is $d_m\beta_m$.

Thus for the case of $(14,10,13)$ code, any failure-pattern with 2 or 3 failures occurring within a single y-section can be repaired efficiently. And for $(14,10,11)$ code, any failure-pattern with 2 or 3 failures occurring in distinct y-sections can be repaired efficiently. In 16, we show the average network traffic needed while repairing 2 failures, assuming that all two-failure-patterns are equally likely. Similar calculation is done for the case of failure-patterns with 1,3 and 4 failures.

Repair Algorithm We present proposed repair algorithm in 1, that is generic for single and multiple erasures. This is invoked whenever savings in bandwidth are possible, i.e, when *is_repair()* returns *true*. In the algorithm, we refer to those non-erased nodes that are not helper nodes as *aloof nodes*.

Algorithm 1 repair

1: Input: $\mathscr{E}$ (erasures), $\mathscr{I}$ (aloof nodes).
2: repair_layers = get_repair_layers($\mathscr{E}$).
3: set $s = 1$.
4: set maxIS $=$ max of IS$(\mathscr{E} \cup \mathscr{I}, z)$ over all z from repair_layers
5: **while** ($1 \leq s \leq$ maxIS)
6: **for** ($z \in$ repair_layers and IS$(\mathscr{E} \cup \mathscr{I}, z) = s$)
7: **if** (IS$(\mathscr{E}, z) > 1$) $G = \phi$
8: **else** {
9: a = the erased node with hole-dot in layer z
10: G is set of all nodes in a's y-section.}
11: $\mathscr{E}' = \mathscr{E} \cup G \cup \mathscr{I}$
12: Compute B sub-chunks in layer z corresponding to all the nodes other than $\mathscr{E}'$
13: Invoke scalar MDS_decode to recover B sub-chunks for all nodes in $\mathscr{E}'$
14: **end for**
15: $s = s + 1$
16: **end while**
17: Compute A chunks corresponding to all the erased nodes, from B chunks in repair layers along with the helper data.

6.1 Evaluation of Multiple Erasures

Network Traffic and Disk Read While the primary benefit of the Clay code is the optimal network traffic and disk read during repair of a single node failure, it also yields savings over RS counterpart code in the case of a large number of mutiple-node failure patterns. To our knowledge, this is the first demonstration of such savings by an MSR code in a multiple-node-failure scenario. We evaluate the performance of codes C4-C6 under W_1 workload injecting multiple node-failures in a setting of 512PGs. The plots for network traffic and disk read, for a single run in the multiple-PG setting is shown in Fig. 17, 16. We emphasize however, that single-run performance of multiple-pg cannot be used to make judgments on performance. The expected behaviour can be seen only if we average out the results over multiple runs.

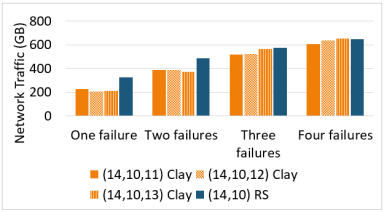


Figure 16: Average network traffic evaluation of C4-C6 against RS codes (W_1 workload, multiple-PG).

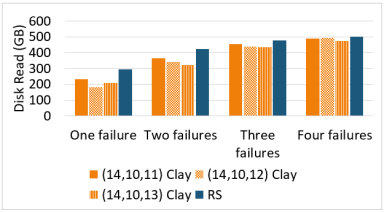


Figure 17: Disk-read evaluation of C4-C6 against RS codes (W_1 workload, multiple-PG).

7 Conclusions

In this work, we present Clay codes that are constructed by placing any MDS code in multiple layers, and performing pair-wise coupling across layers. Clay codes provide the first practical implementation of an MSR code that offers (a) low storage overhead, (b) optimal repair bandwidth, (c) low sub-packetization level, (d) support for both single and multiple-node repairs and (e) uniform repair performance of data and parity nodes while permitting faster & more efficient repair. We implemented Clay codes in Ceph and evaluated the repair performance of six example codes, whose parameters match with known erasure-code deployments in practice. A particular Clay example code, with storage overhead 1.25, is shown to reduce repair network traffic, disk read and repair times by factors of 2.9, 3.4 and 3 respectively. We also modified Ceph to support any vector code, and our contribution is on its way to Ceph's master codebase. In summary, we feel that Clay codes are good candidates to make the transition from theory to practice.

References

[1] Backblaze data service provider. https://www.backblaze.com/blog/reed-solomon/. Accessed: 2017-Sep-28.

[2] Ceph source code (master branch). https://github.com/ceph/ceph.

[3] Red hat ceph storage: Scalable object storage on qct servers - *a performance and sizing guide*. Reference Architecture.

[4] Tutorial: Erasure coding for storage applications. http://web.eecs.utk.edu/~plank/plank/papers/FAST-2013-Tutorial.html. Accessed: 2017-Sep-28.

[5] BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Computers 44*, 2 (1995), 192–202.

[6] CHEN, H. C., HU, Y., LEE, P. P., AND TANG, Y. Nccloud: A network-coding-based storage system in a cloud-of-clouds. *IEEE Transactions on Computers 63*, 1 (2013), 31–44.

[7] CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), pp. 1–14.

[8] DIMAKIS, A., GODFREY, P., WU, Y., WAINWRIGHT, M., AND RAMCHANDRAN, K. Network coding for distributed storage systems. *IEEE Transactions on Information Theory 56*, 9 (Sep. 2010), 4539–4551.

[9] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation* (Vancouver, BC, 2010), USENIX.

[10] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003* (2003), pp. 29–43.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[12] GLIGOROSKI, D., KRALEVSKA, K., JENSEN, R. E., AND SIMONSEN, P. Locally repairable and locally regenerating codes obtained by parity-splitting of hashtag codes. *CoRR abs/1701.06664* (2017).

[13] HU, Y., CHEN, H., LEE, P., AND TANG, Y. NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of the 10thth USENIX Conference on File and Storage Technologies(FAST)* (2012).

[14] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 15–26.

[15] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC.

[16] JIANG, W., HU, C., ZHOU, Y., AND KANEVSKY, A. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage 4*, 3 (Nov. 2008), 7:1–7:25.

[17] KHAN, O., BURNS, R. C., PLANK, J. S., AND HUANG, C. In Search of I/O-Optimal Recovery from Disk Failures. In *USENIX HotStorage* (2011).

[18] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., TANG, L., AND KUMAR, S. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 383–398.

[19] PAMIES-JUAREZ, L., BLAGOJEVIC, F., MATEESCU, R., GUYOT, C., GAD, E. E., AND BANDIC, Z. Opening the chrysalis: On the real repair performance of MSR codes. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies* (2016), pp. 81–94.

[20] PLANK, J., GREENAN, K., MILLER, E., AND HOUSTON, W. Gf-complete: A comprehensive open source library for galois field arithmetic. *University of Tennessee, Tech. Rep. UT-CS-13-703* (2013).

[21] PLANK, J. S., AND GREENAN, K. M. Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0. Tech. rep., Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[22] RASHMI, K. V., CHOWDHURY, M., KOSAIAN, J., STOICA, I., AND RAMCHANDRAN, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* (2016), pp. 401–417.

[23] RASHMI, K. V., NAKKIRAN, P., WANG, J., SHAH, N. B., AND RAMCHANDRAN, K. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST,* (2015), pp. 81–94.

[24] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'13, 2013* (2013), USENIX Association.

[25] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *ACM SIGCOMM 2014 Conference,* (2014), pp. 331–342.

[26] RASHMI, K. V., SHAH, N. B., AND KUMAR, P. V. 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 (Aug 2011), 5227–5239.

[27] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D. S., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. *PVLDB 6*, 5 (2013), 325–336.

[28] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), FAST '07, USENIX Association.

[29] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems*

Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA (2006), pp. 307–320.

[30] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. Grid resource management - CRUSH: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA* (2006), p. 122.

[31] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07), November 11, 2007, Reno, Nevada, USA* (2007), pp. 35–44.

[32] YE, M., AND BARG, A. Explicit constructions of optimal-access MDS codes with nearly optimal sub-packetization. *IEEE Trans. Information Theory 63*, 10 (2017), 6307–6317.