Clay Codes: Moulding MDS Codes to Yield an MSR Code

Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan
Indian Institute of Science (IISc)

P. Vijay Kumar (IISc and USC)

Alexander Barg, Min Ye (UMD)

Srinivasan Narayanamurthy, Syed Hussain, Siddhartha Nandi (NetApp)

16th USENIX Conference on File and Storage Technologies (FAST), 2018
Oakland, CA

Erasure Coding for Fault Tolerance

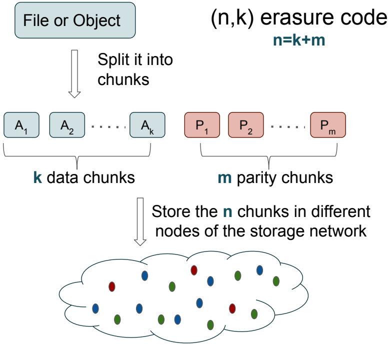
- Fault tolerance is key to making data loss a very remote possibility

Erasure Coding for Fault Tolerance

- Fault tolerance is key to making data loss a very remote possibility
- Fault tolerance is achieved using erasure coding

Erasure Coding for Fault Tolerance

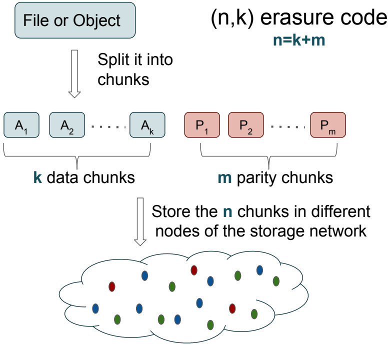
- Fault tolerance is key to making data loss a very remote possibility
- Fault tolerance is achieved using erasure coding



The *n* chunks taken together, form a stripe.

Erasure Coding for Fault Tolerance

- Fault tolerance is key to making data loss a very remote possibility
- Fault tolerance is achieved using erasure coding



The n chunks taken together, form a stripe.

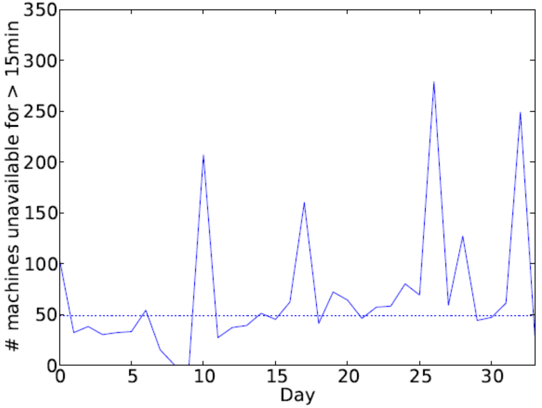
Two Key Performance Measures

1. Storage Overhead $\frac{n}{k}$
2. Fault Tolerance - at most m storage units

MDS Codes

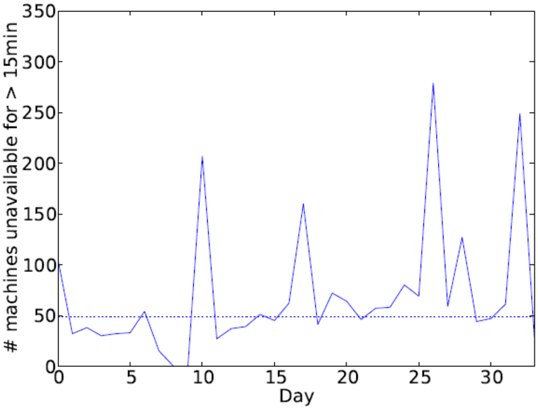
1. For given (n, k), **MDS erasure codes** have the maximum-possible fault tolerance
2. RAID 6 and Reed-Solomon codes are examples of MDS codes.

Erasure Codes and Node Failures



- A median of 50 nodes are unavailable per day.
- 98% of the failures are single node failures.
- A median of 180TB of network traffic per day is generated in order to reconstruct the RS coded data corresponding to unavailable machines.

Erasure Codes and Node Failures



- A median of 50 nodes are unavailable per day.
- 98% of the failures are single node failures.
- A median of 180TB of network traffic per day is generated in order to reconstruct the RS coded data corresponding to unavailable machines.
- **Thus there is a strong need for erasure codes that can efficiently recover from single-node failures.**

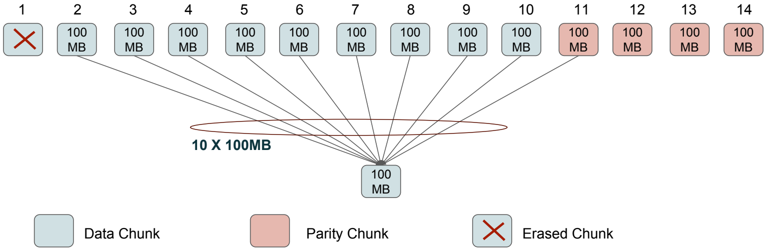
Image courtesy: Rashmi et al.: "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," USENIX Hotstorage, 2013.

Conventional Node Repair of an RS Code

The conventional repair of an RS code is inefficient

Conventional Node Repair of an RS Code

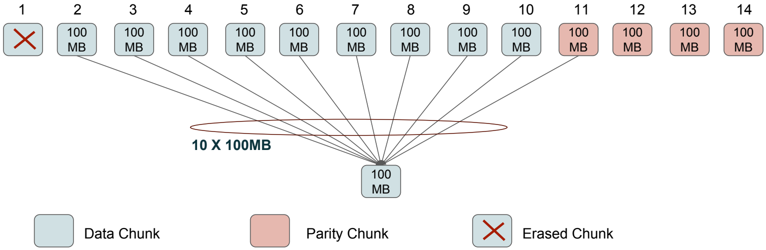
The conventional repair of an RS code is inefficient



In the example $(14, 10)$ RS code,

Conventional Node Repair of an RS Code

The conventional repair of an RS code is inefficient

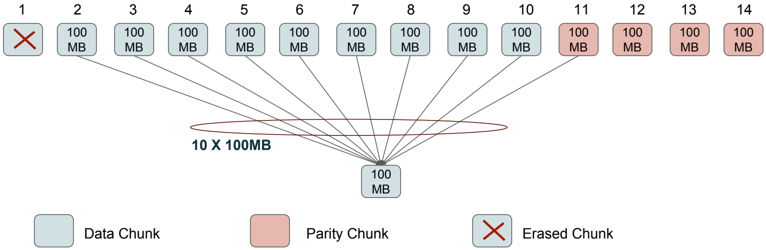


In the example $(14, 10)$ RS code,

1. the amount of data downloaded to repair 100MB of data equals 1GB.

Conventional Node Repair of an RS Code

The conventional repair of an RS code is inefficient



In the example $(14, 10)$ RS code,

1. the amount of data downloaded to repair 100MB of data equals 1GB.

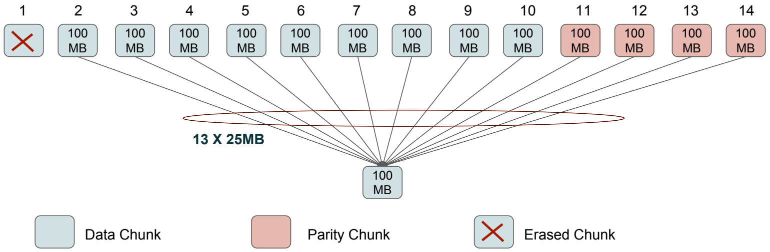
clearly, there is room for improvement...

Regenerating Codes

1. We will deal here only in the subclass of regenerating codes known as Minimum Storage Regeneration (MSR) codes
2. MSR codes are MDS and have least possible repair bandwidth
3. Repair bandwidth is defined as the total amount of data downloaded for repair of a single node

Regenerating Codes

1. We will deal here only in the subclass of regenerating codes known as Minimum Storage Regeneration (MSR) codes
2. MSR codes are MDS and have least possible repair bandwidth
3. Repair bandwidth is defined as the total amount of data downloaded for repair of a single node

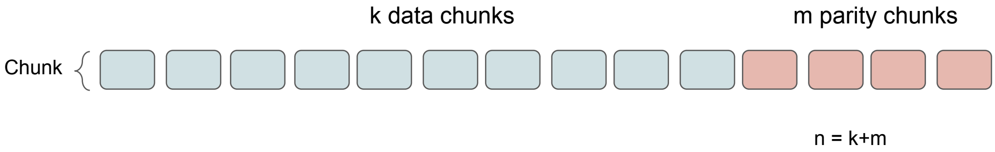


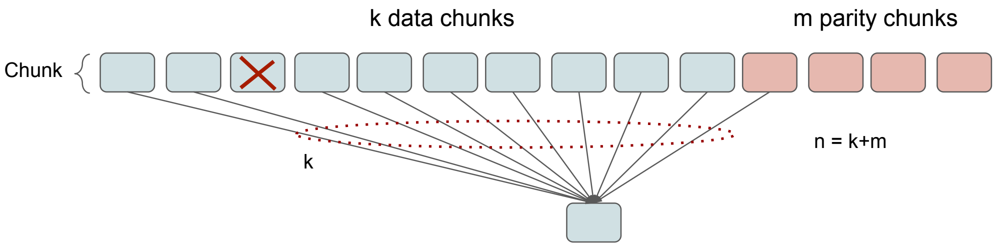
1. Size of failed node's contents: 100MB
2. RS repair BW: 1 GB
3. MSR Repair BW: 325 MB

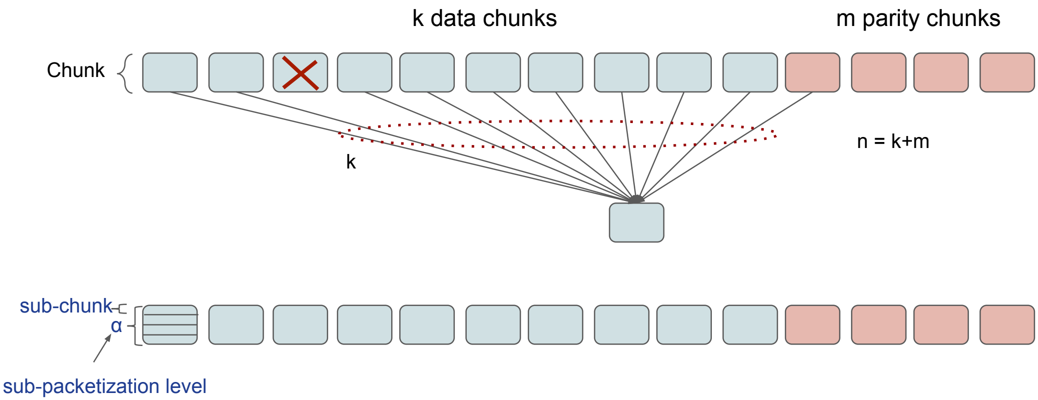
Key to the Impressive, Low-Repair BW of MSR Codes

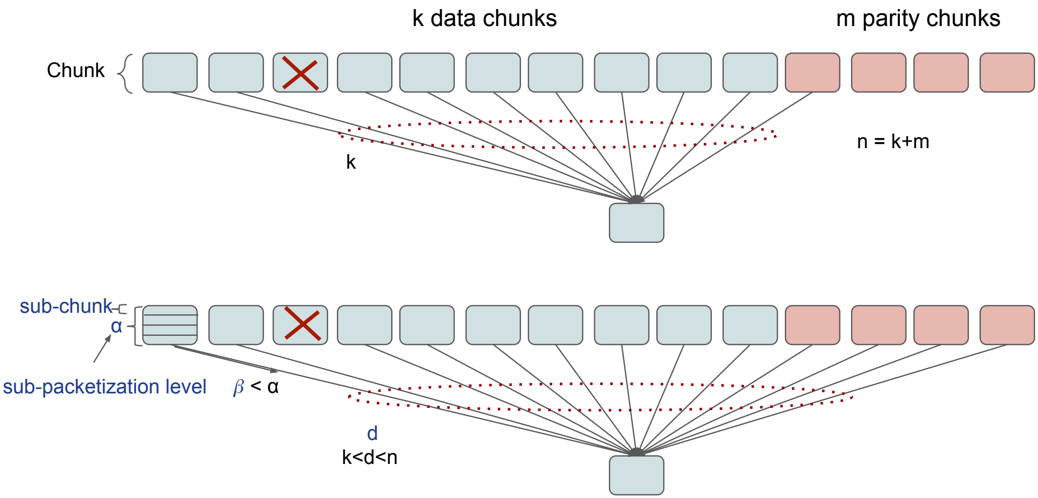
Key to the Impressive, Low-Repair BW of MSR Codes

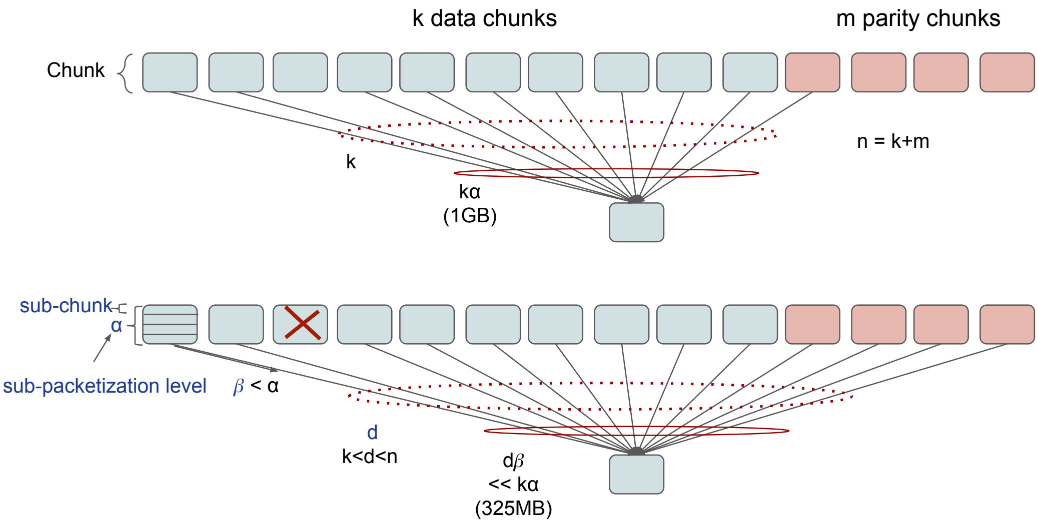
In a nutshell: sub-packetization... we explain...


k data chunks
m parity chunks
Chunk
n = k+m

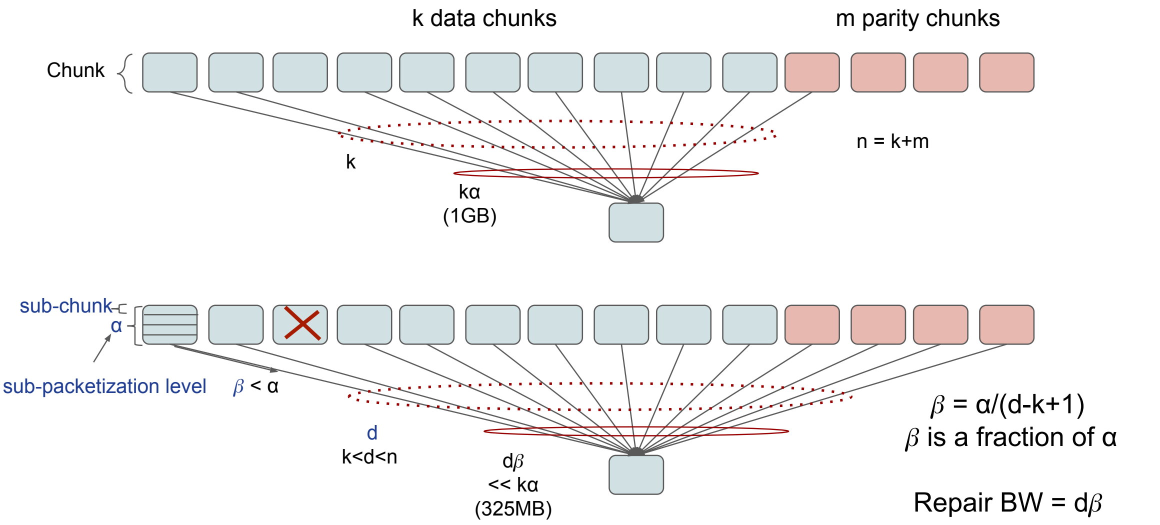

k data chunks
m parity chunks
Chunk
n = k+m
k


k data chunks
m parity chunks
Chunk
k
n = k+m
sub-chunk
α
sub-packetization level

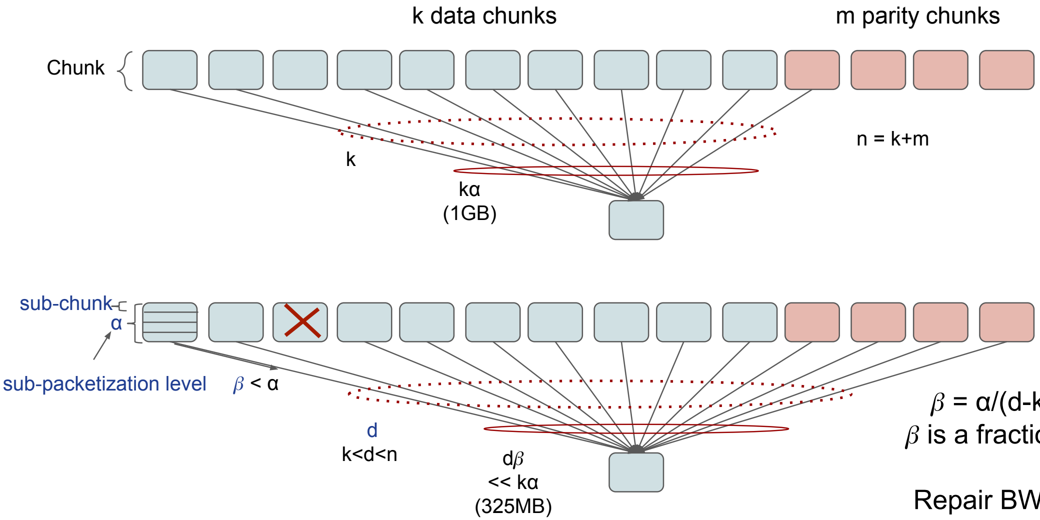

k data chunks
m parity chunks
Chunk
n = k+m
k
sub-chunk
α
sub-packetization level
β < α
d
k<d<n



k data chunks
m parity chunks
Chunk
n = k+m
k
kα
(1GB)
sub-chunk
α
sub-packetization level
β < α
d
k<d<n
dβ
<< kα
(325MB)



k data chunks
m parity chunks
Chunk
n = k+m
k
kα
(1GB)
sub-chunk
α
sub-packetization level
β < α
d
k<d<n
dβ
<< kα
(325MB)
β = α/(d-k+1)
β is a fraction of α
Repair BW = dβ
We consider d=n-1, then
Repair BW = (n-1)α/(n-k)



$\beta = \alpha/(d-k+1)$

β is a fraction of α

Repair BW = $d\beta$

We consider d=n-1, then

Repair BW = (n-1)α/(n-k)

Larger the m=n-k, larger the savings!!

Additional Properties Desired of an MSR Code

1. Minimal Disk Read (IO Optimality): Read exactly what is needed to be transferred

Additional Properties Desired of an MSR Code

1. Minimal Disk Read (IO Optimality): Read exactly what is needed to be transferred
2. Minimize sub-packetization level α

Additional Properties Desired of an MSR Code

1. Minimal Disk Read (IO Optimality): Read exactly what is needed to be transferred
2. Minimize sub-packetization level α
   - sub-chunk size $= \frac{\text{chunk size}}{\alpha} = N$ bytes.
   - During repair, β sub-chunks are read.

Additional Properties Desired of an MSR Code

1. Minimal Disk Read (IO Optimality): Read exactly what is needed to be transferred

2. Minimize sub-packetization level α
   - sub-chunk size $= \frac{\text{chunk size}}{\alpha} = N$ bytes.
   - During repair, β sub-chunks are read.
   - If sub-chunks are not contiguous, only N bytes are read sequentially.
   - Smaller the α better the sequentiality!!

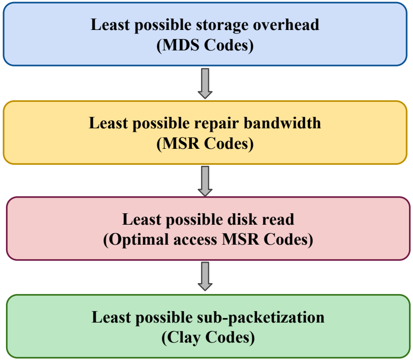
Additional Properties Desired of an MSR Code

1. Minimal Disk Read (IO Optimality): Read exactly what is needed to be transferred

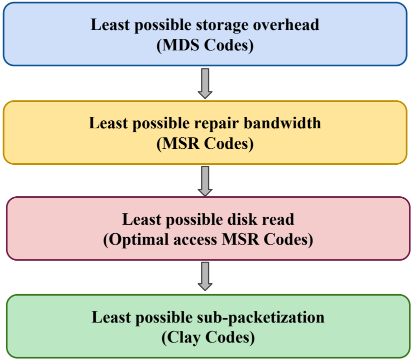
2. Minimize sub-packetization level α
   - sub-chunk size $= \frac{\text{chunk size}}{\alpha} = N$ bytes.
   - During repair, β sub-chunks are read.
   - If sub-chunks are not contiguous, only N bytes are read sequentially.
   - Smaller the α better the sequentiality!!

3. Small field size, low-complexity implementation.

4-way Optimality of Clay code



4-way Optimality of Clay code



among the class of MSR codes, the Clay code is arguably a champion...

Image courtesy: denverpost.com

Placing the Clay Code in Perspective

Comparing the Clay code with repair-efficient codes that have undergone systems implementation

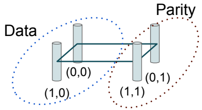
Code	MDS	Least Repair BW	Least Disk Read	Least α	Restrictions	Implemented Distributed Systems
Piggybacked RS (Sigcomm 2014)	✔	✗	✗	-	None	HDFS
Product Matrix (FAST 2015)	✔	✔	✔	✔	**Limited to Storage Overhead > 2**	Own System
Butterfly Code (FAST 2016)	✔	✔	✗	✗	Limited to the 2 parity nodes	HDFS, Ceph
HashTag Code (Trans. on Big Data 2017)	✔	✗	✗	-	Only systematic node repair	HDFS
Clay (FAST 2018)	✔	✔	✔	✔	**None!**	Ceph

- The Butterfly, HashTag codes have least disk read for systematic node repair.

#HT: A similar table given in the paper and the poster had erroneous information on HT codes.

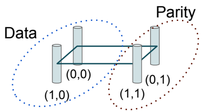
Clay Code Construction

Moulding an MDS Code to Yield a $(4, 2)$ Clay Code



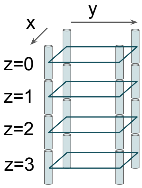
Two sub-chunks are encoded using $(4, 2)$ scalar MDS code.

Moulding an MDS Code to Yield a $(4, 2)$ Clay Code



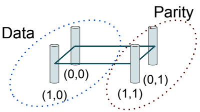
Two sub-chunks are encoded using $(4, 2)$ scalar MDS code.

$\rightarrow$



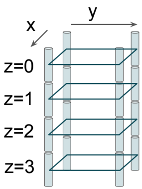
Layer four such units.

Moulding an MDS Code to Yield a $(4, 2)$ Clay Code



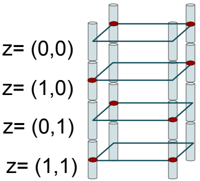
Two sub-chunks are encoded using $(4, 2)$ scalar MDS code.

$\rightarrow$



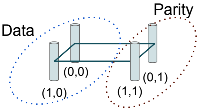
Layer four such units.

$\rightarrow$



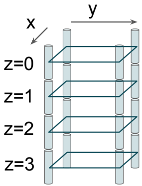
Index each layer z using two bits (corresponding to the location of the two red dots in that layer).

Moulding an MDS Code to Yield a $(4, 2)$ Clay Code



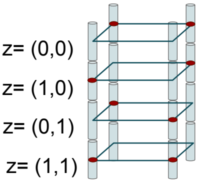
Two sub-chunks are encoded using $(4, 2)$ scalar MDS code.

→

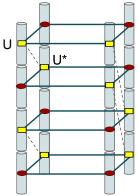


Layer four such units.

→

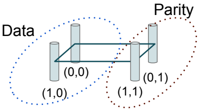


Index each layer z using two bits (corresponding to the location of the two red dots in that layer).



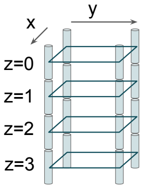
sub-chunks such as (U, U^*) are paired (yellow rectangles connected by a dotted line).

Moulding an MDS Code to Yield a (4, 2) Clay Code



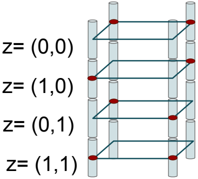
Two sub-chunks are encoded using $(4, 2)$ scalar MDS code.

→

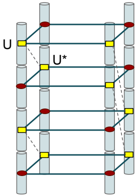


Layer four such units.

→



Index each layer z using two bits (corresponding to the location of the two red dots in that layer).



sub-chunks such as (U, U^*) are paired (yellow rectangles connected by a dotted line).

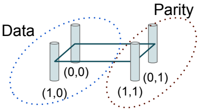
→

Pairwise Forward Transform (PFT)

$$\begin{bmatrix} C \\ C^* \end{bmatrix} = A \begin{bmatrix} U \\ U^* \end{bmatrix}$$

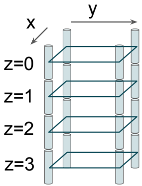
Any two sub-chunks out of $\{U, U^*, C, C^*\}$ can be computed from remaining two.

Moulding an MDS Code to Yield a (4, 2) Clay Code



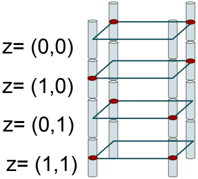
Two sub-chunks are encoded using (4, 2) scalar MDS code.

→

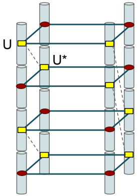


Layer four such units.

→

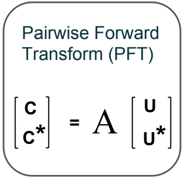


Index each layer z using two bits (corresponding to the location of the two red dots in that layer).



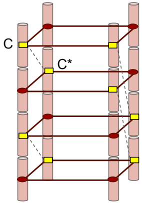
sub-chunks such as (U, U^*) are paired (yellow rectangles connected by a dotted line).

→



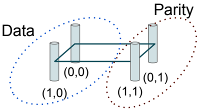
Any two sub-chunks out of $\{U, U^*, C, C^*\}$ can be computed from remaining two.

→



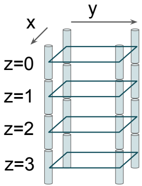
Perform PFT on paired sub-chunks and copy the unpaired sub-chunks to get the Clay code.

Moulding an MDS Code to Yield a $(4, 2)$ Clay Code



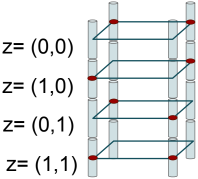
Two sub-chunks are encoded using $(4, 2)$ scalar MDS code.

→

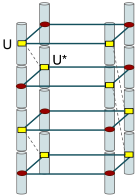


Layer four such units.

→



Index each layer z using two bits (corresponding to the location of the two red dots in that layer).



sub-chunks such as (U, U^*) are paired (yellow rectangles connected by a dotted line).

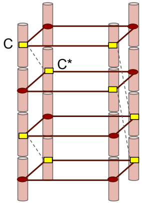
→

Pairwise Forward Transform (PFT)

$$\begin{bmatrix} C \\ C^* \end{bmatrix} = A \begin{bmatrix} U \\ U^* \end{bmatrix}$$

Any two sub-chunks out of $\{U, U^*, C, C^*\}$ can be computed from remaining two.

→



Perform PFT on paired sub-chunks and copy the unpaired sub-chunks to get the Clay code.

Can be generalized to any (n, k, d)!!

Encoding the Clay Code

- The previous slide did not explain how encoding takes place as the code was not in systematic form.
- We will now explain encoding data under the Clay Code.

Consider a file of size 64MB

64MB

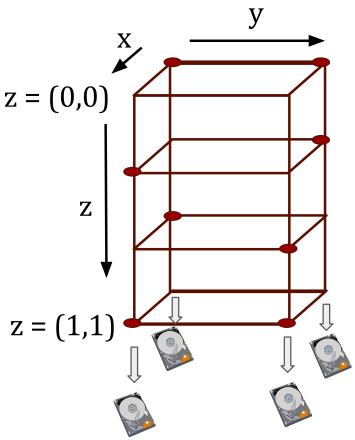
- We show encoding of the file using (n = 4, k = 2) Clay code.

Break the file into k = 2 data chunks each of 32MB.

32MB	32MB

3D cube representation of Clay Code

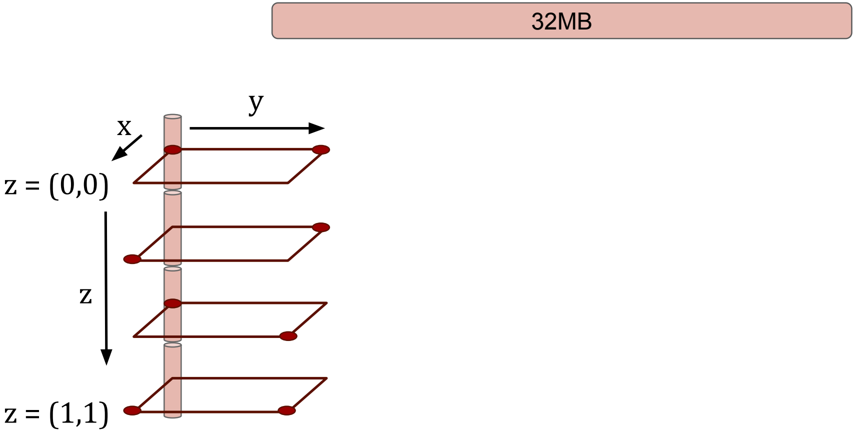
32MB	32MB



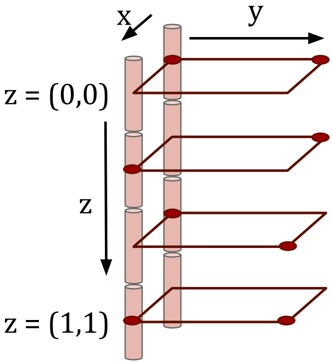
The cube has:

- 4 columns, which correspond to the 4 chunks (each of size 32MB, stored in a different disk/node).
- 4 horizontal planes.
- Each column has 4 points that correspond to sub-chunks of size 8MB

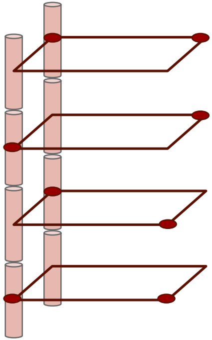
Place two 32MB chunks in two data nodes



Place two 32MB chunks in two data nodes

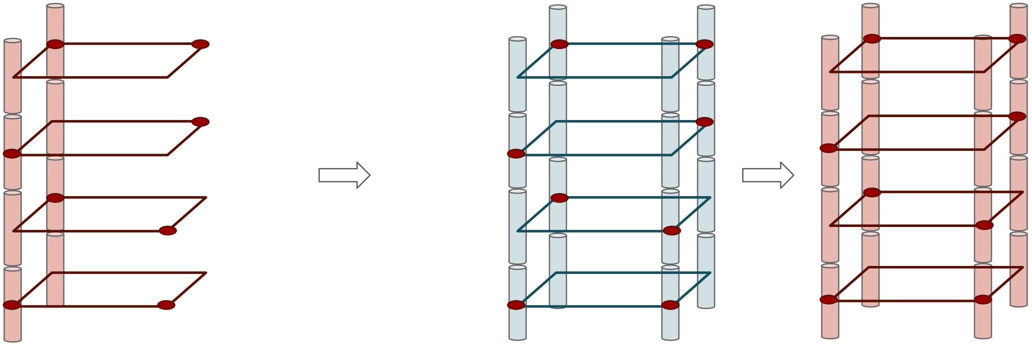


We now have the data nodes

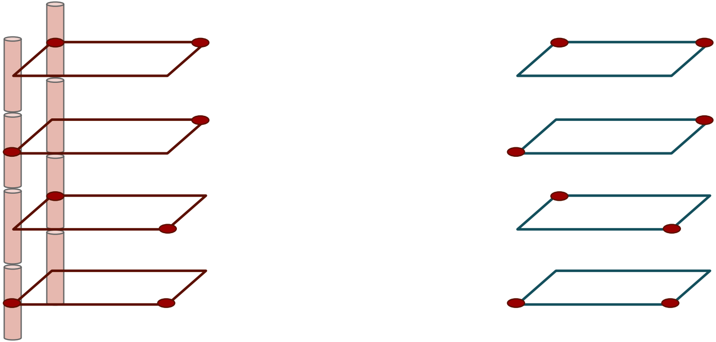


We will now compute the parity nodes

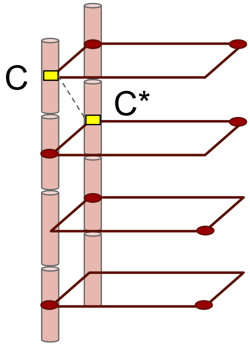
Will get there through an intermediate “Uncoupled data cube”

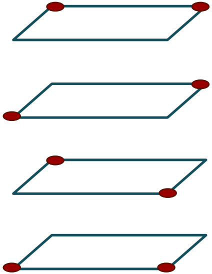


Start filling the Uncoupled data cube on the right as follows

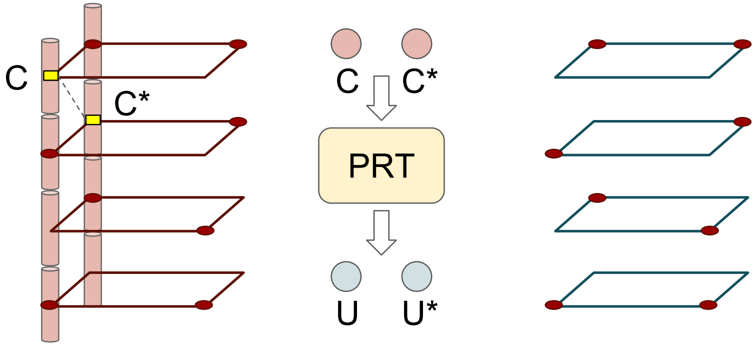


Certain pairs of points in the cube are "coupled"

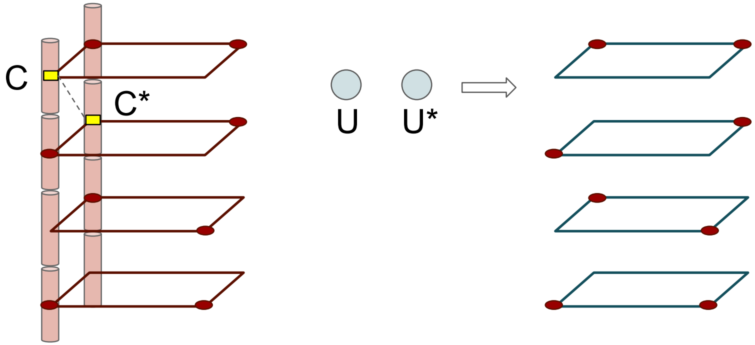




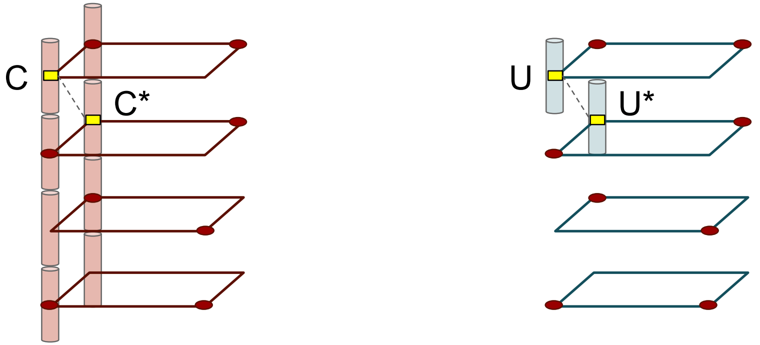
PRT is a 2x2 matrix transform, It is reverse of PFT



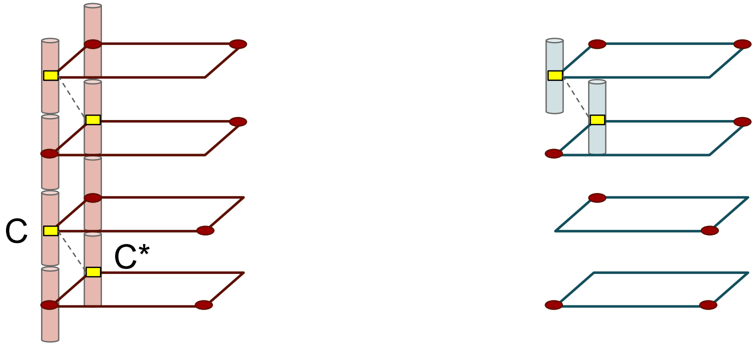
Place the sub-chunks obtained in the uncoupled data cube



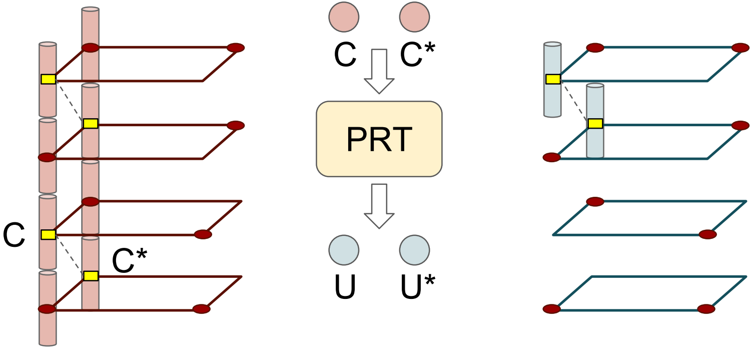
Place the sub-chunks obtained in the uncoupled data cube



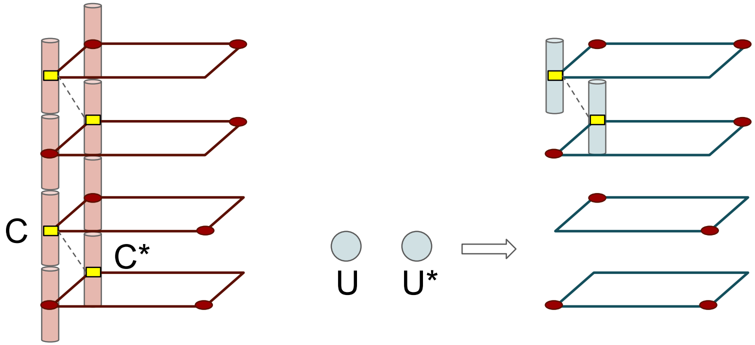
Place the sub-chunks obtained in the uncoupled data cube



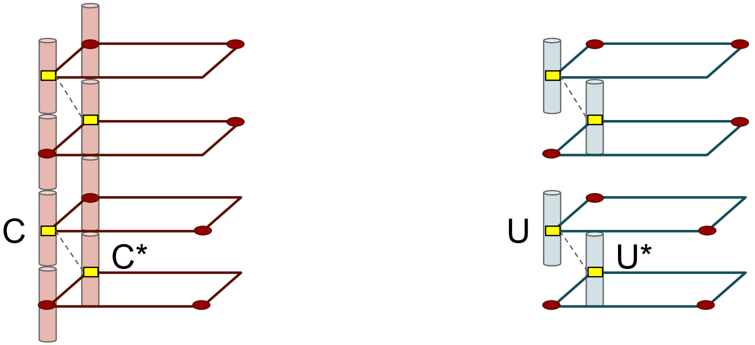
Place the sub-chunks obtained in the uncoupled data cube



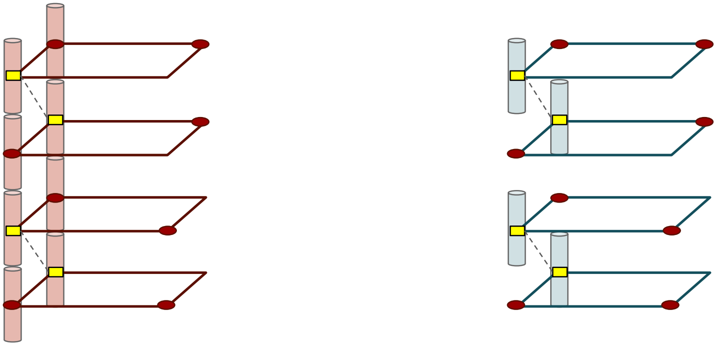
Place the sub-chunks obtained in the uncoupled data cube



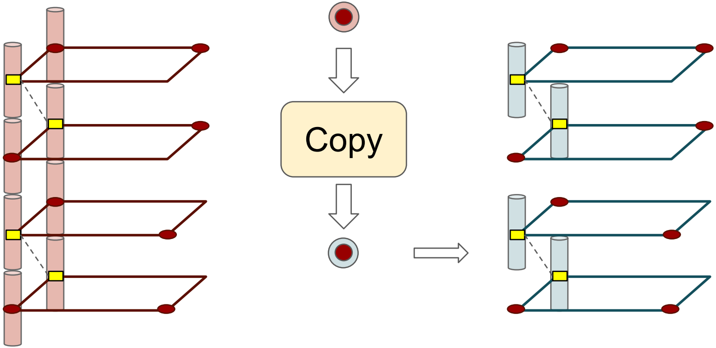
Place the sub-chunks obtained in the uncoupled data cube



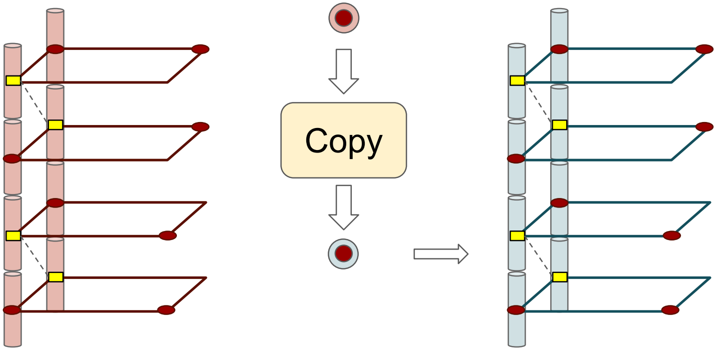
Place the sub-chunks obtained in the uncoupled data cube



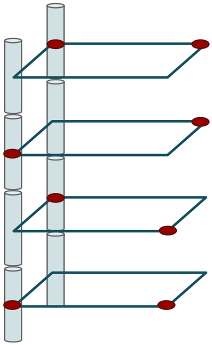
Red dotted sub-chunks are not paired, they are simply carried over



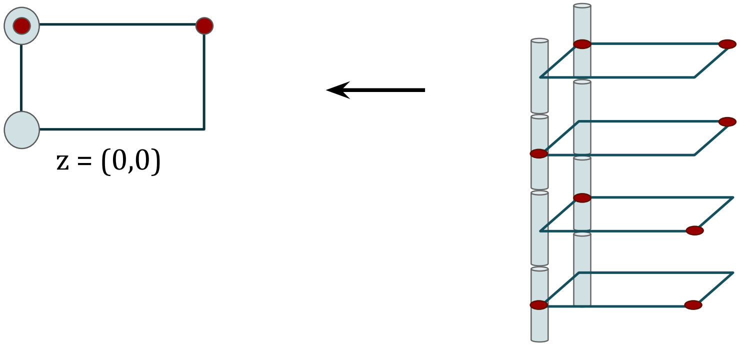
Red dotted sub-chunks are not paired, they are simply carried over



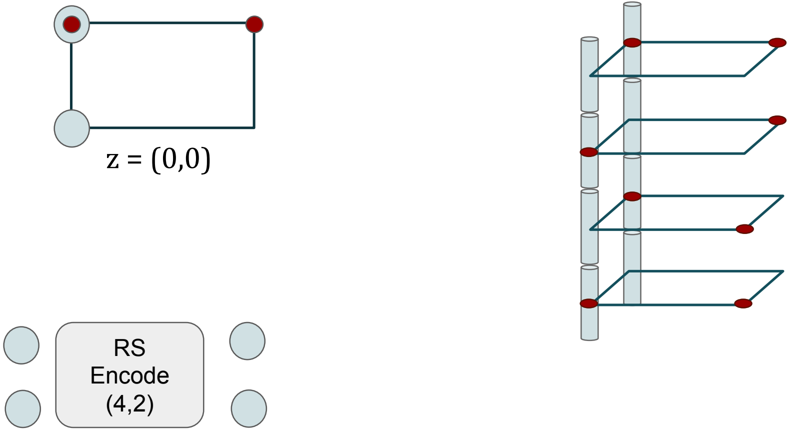
We now have data-part of the uncoupled data cube



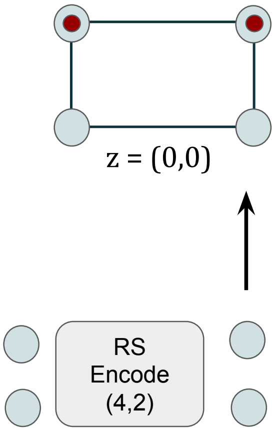
Each plane is Reed-Solomon encoded to obtain parity points (sub-chunks)

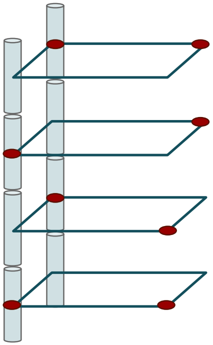


Each plane is Reed-Solomon encoded to obtain parity points (sub-chunks)

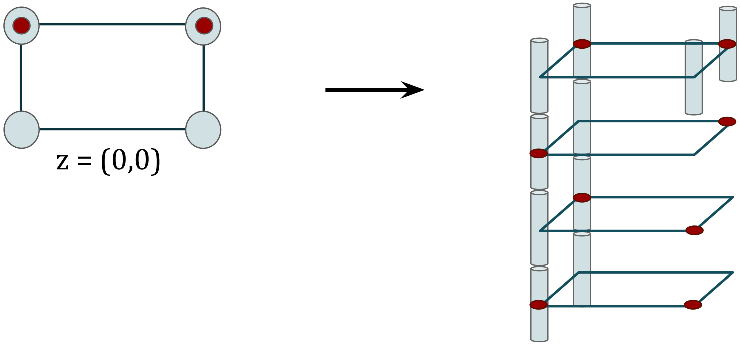


Each plane is Reed-Solomon encoded to obtain parity points (sub-chunks)

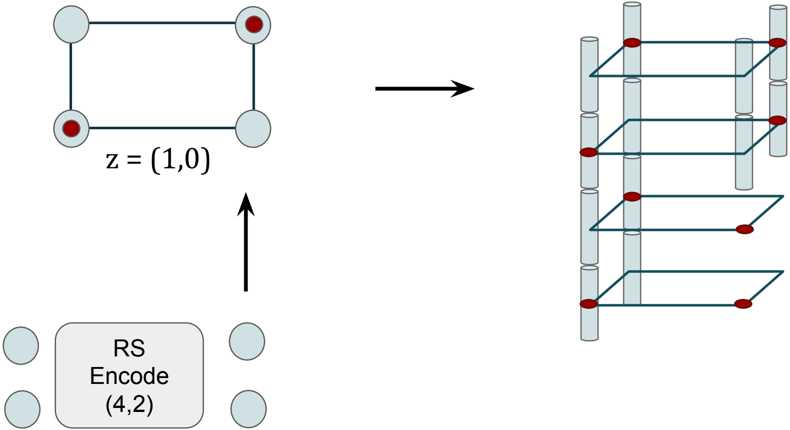




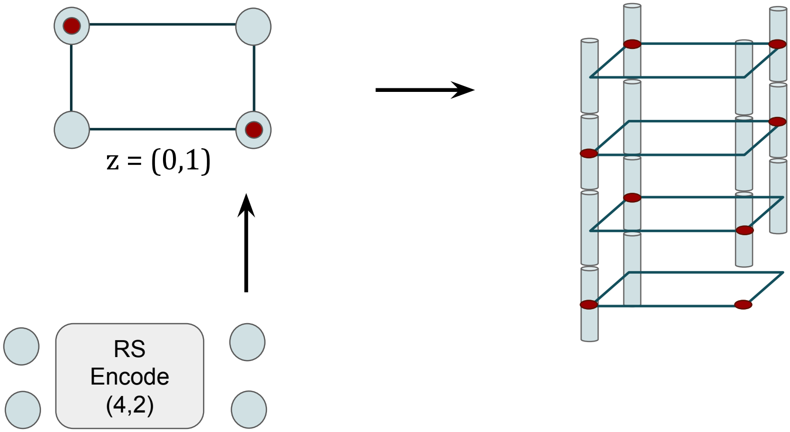
Each plane is Reed-Solomon encoded to obtain parity points (sub-chunks)



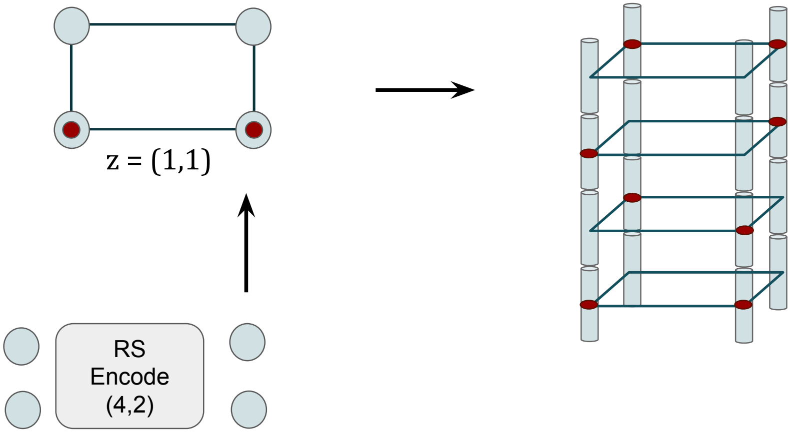
Each plane is Reed-Solomon encoded to obtain parity points (sub-chunks)



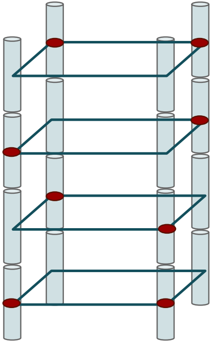
Each plane is Reed-Solomon encoded to obtain parity points (sub-chunks)



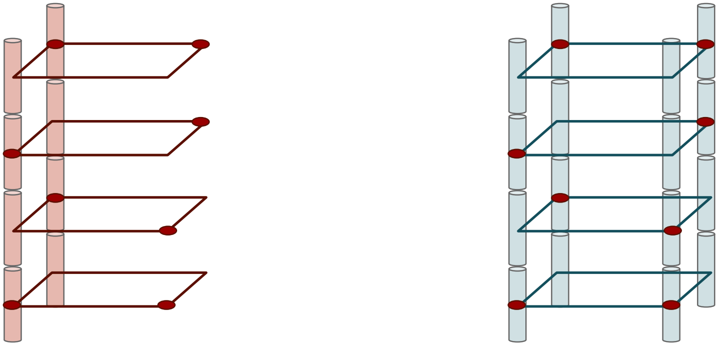
Each plane is Reed-Solomon encoded to obtain parity points (sub-chunks)



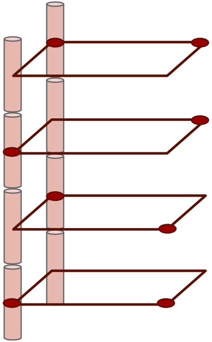
Now we have the complete Uncoupled data cube

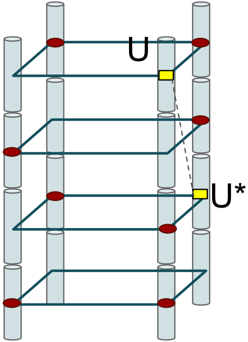


Parity sub-chunks of Coupled data cube can now be computed

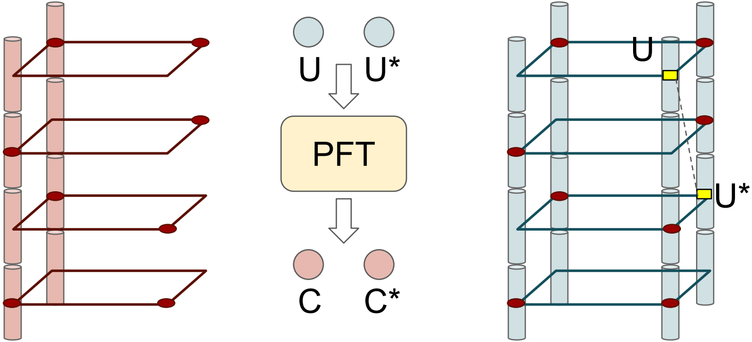


Perform PFT

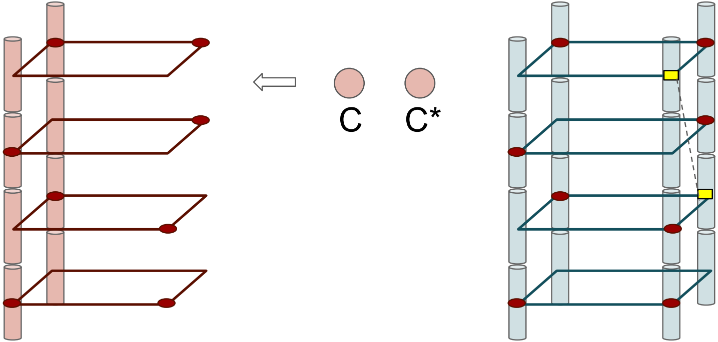




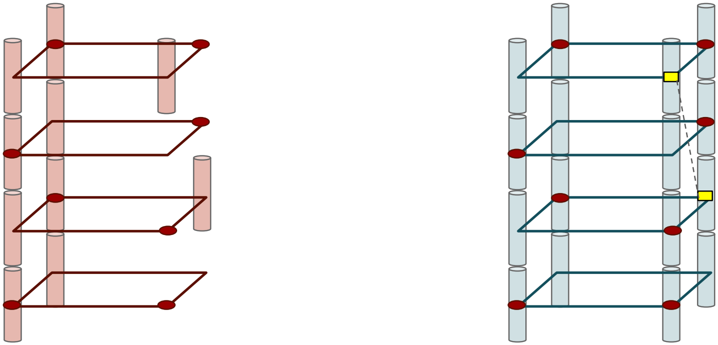
Perform PFT



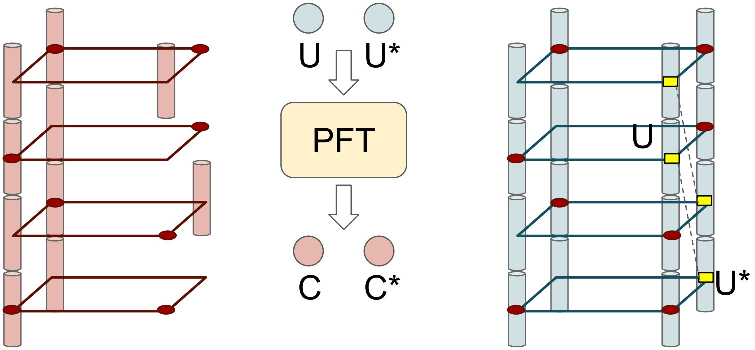
Perform PFT



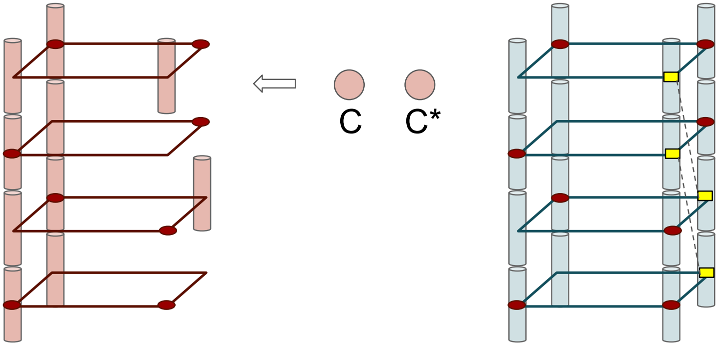
Perform PFT



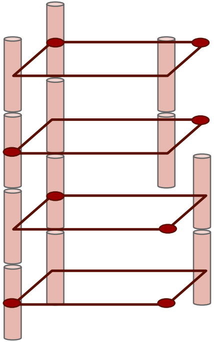
Perform PFT

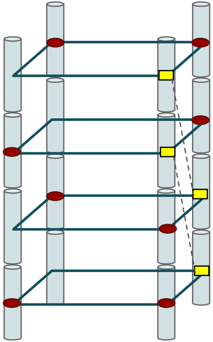


Perform PFT

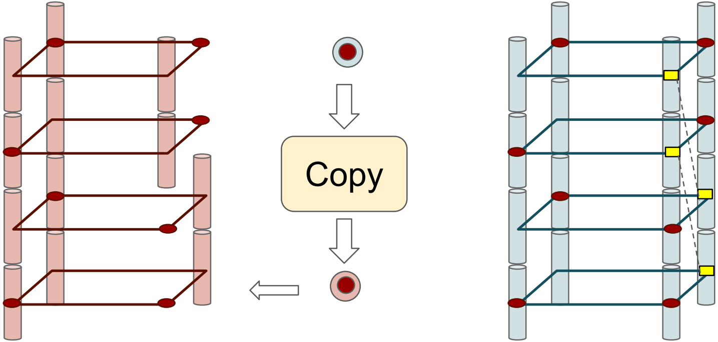


Perform PFT

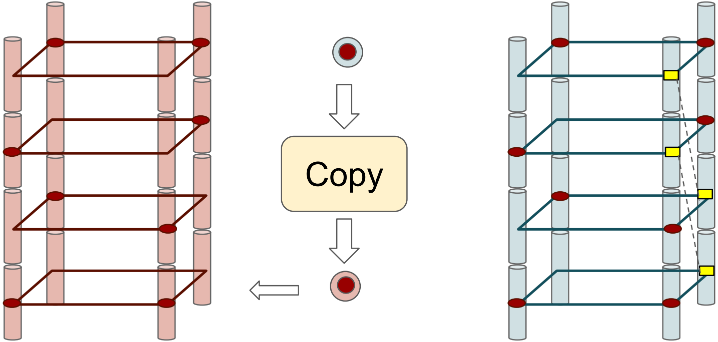




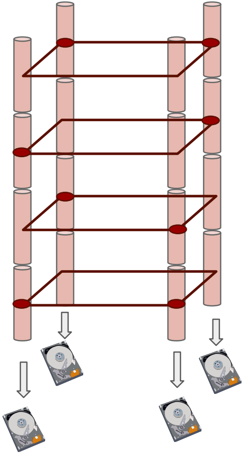
Red dotted sub-chunks are simply carried over



Red dotted sub-chunks are simply carried over

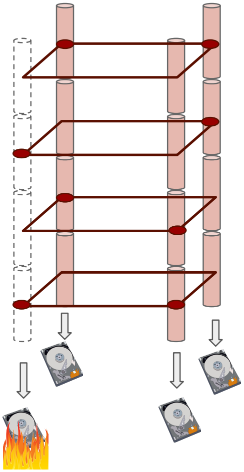


The encoding is now complete!

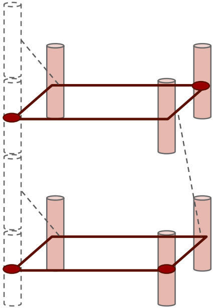


Recovery from single node failure

Node Repair: One node fails

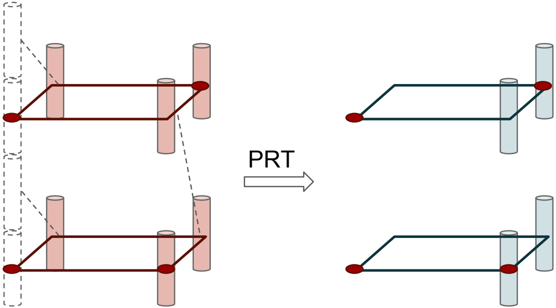


Only half of planes participate in repair

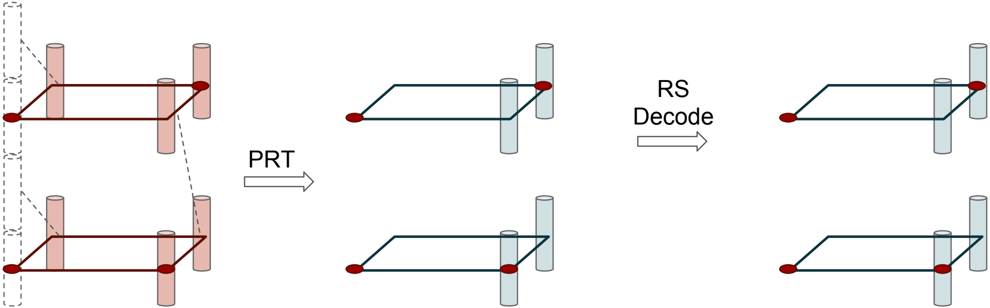


- Total Helper Data = 8MB X 3 X 2 = 48MB
- As opposed to RS code = 8MB X 2 X 4 = 64MB
- Much larger savings seen for m > 2

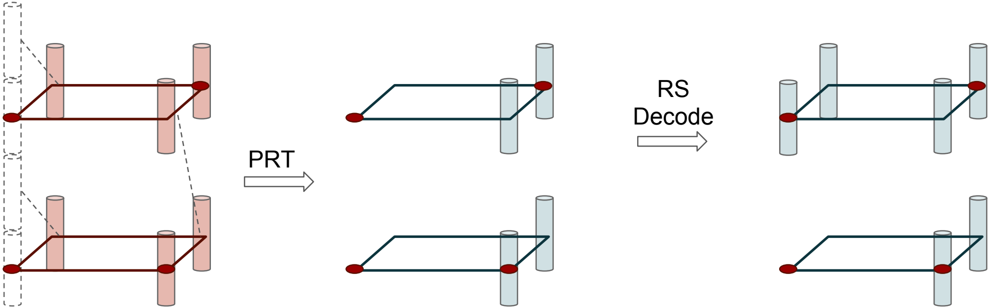
Perform PRT to get possible uncoupled sub-chunks



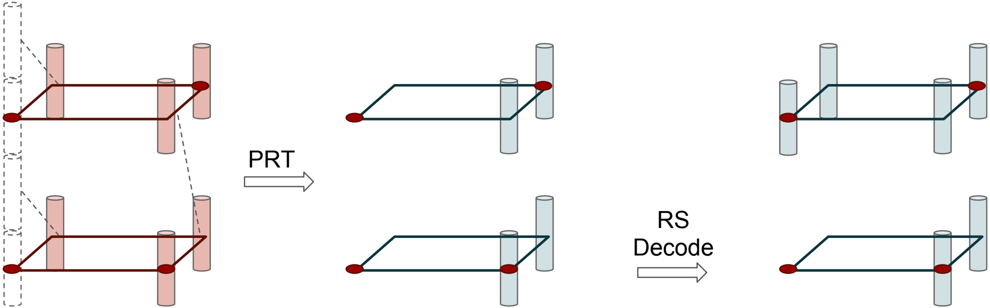
Run RS decoding on each of the selected planes



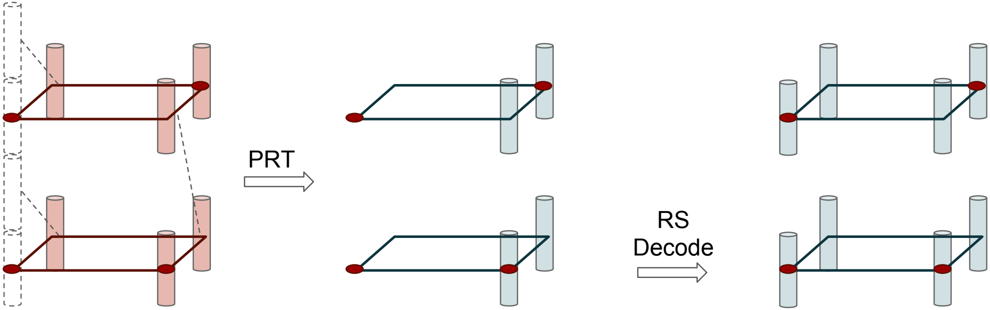
Run RS decoding on each of the selected planes



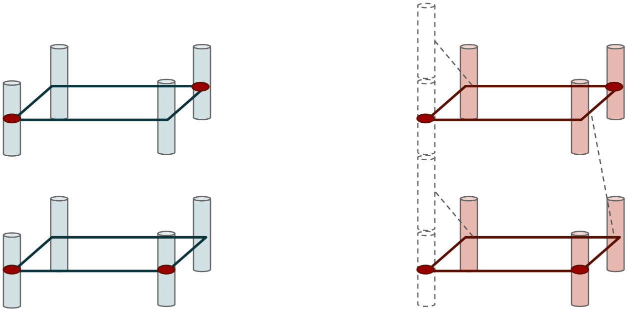
Run RS decoding on each of the selected planes



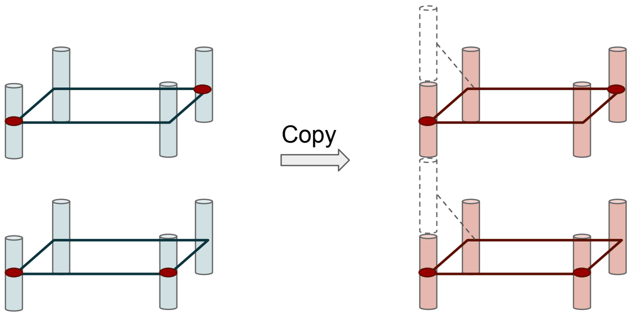
Run RS decoding on each of the selected planes



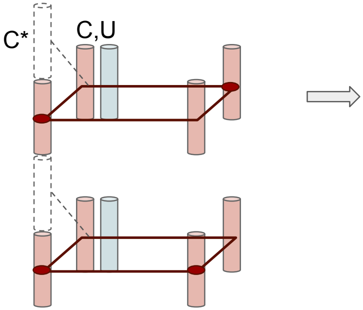
We now have the following sub-chunks available



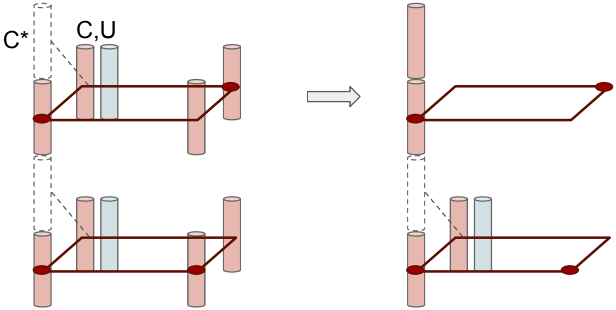
Half the number of required sub-chunks are now already computed



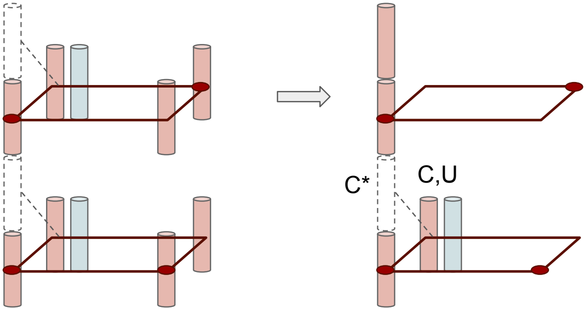
Compute C* from C and U



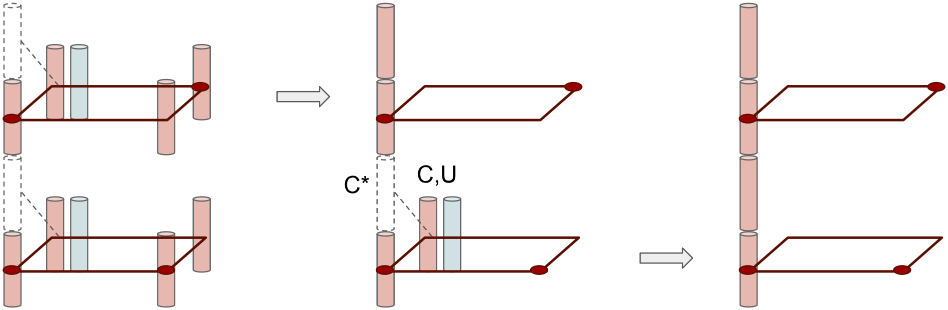
Compute C* from C and U



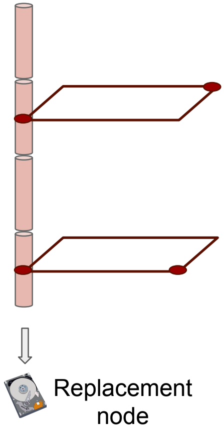
Compute C* from C and U



Compute C* from C and U



Content of failed node is now completely recovered



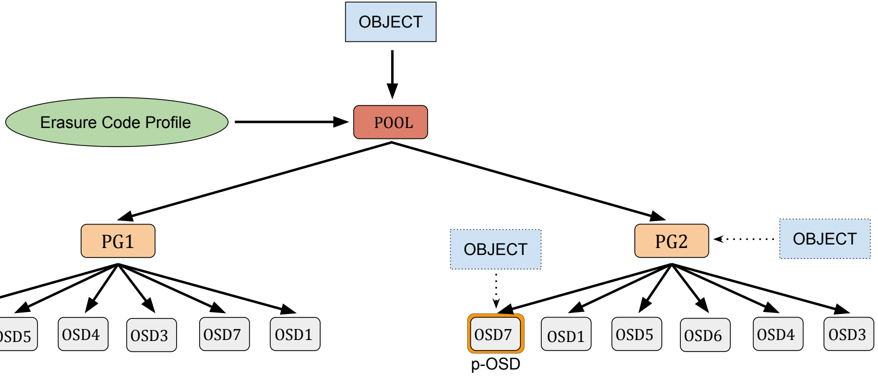
MDS Property of Clay Code

- Any $n-k$ node failures can be recovered from.
- The decoding algorithm recovers the lost symbols layer by layer sequentially.
- It uses functions scalar MDS decode, PFT, PRT and the function that computes U from $\{U^*, C\}$.
- Decoding algorithm involves α scalar MDS decode operations along with $2n\beta$ Galois field scalar multiplications and $n\beta$ Galois XOR operations.
- RS decode for the same amount of data involve α scalar MDS decode operations.

Implementation and Evaluation of Clay Code

Ceph: Architecture

- Object Storage Daemon (OSD): process of Ceph, associated with a storage unit.
- Pool: Logical partitions, associated with an erasure-code profile.
- Placement Group(PG): Collection of n OSDs.
- Each pool can have a single or multiple PGs associated with it.



Ceph: Contributions

- We introduced the notion of sub-chunking to enable use of vector erasure codes with Ceph.

  osd: introduce sub-chunks to erasure code plugin interface #15193

  Merged tchaikov merged 3 commits into `ceph:master` from `mynaramana:arraycode` on Nov 1, 2017

  It is now part of Ceph's master codebase :)

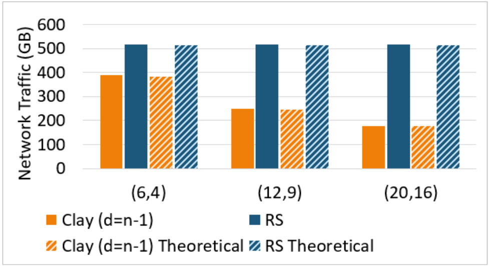
- Clay code will soon be available as an erasure code plugin [1] in Ceph for all parameters (n, k, d)

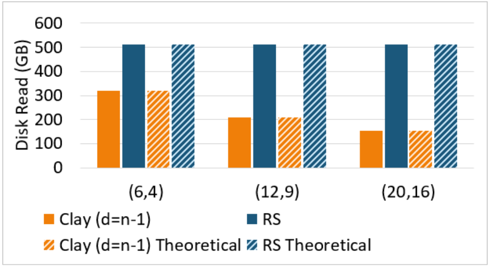
[1] https://github.com/ceph/ceph/pull/14300

Evaluation of the Clay Code

- Evaluated on a 26 node (m4.xlarge) AWS cluster.
- One node hosts Monitor (MON) process of Ceph.
- Remaining 25 nodes host one OSD each.
- Each node has 500GB SSD type volume attached.
- Two workloads
  - Workload W1: fixed size 64MB objects $\rightarrow$ stripe size 64MB
  - Workload W2: mixture of 1MB, 32MB, and 64MB size objects, $\rightarrow$ stripe size 1MB
- Both single PG and multiple PG (512 PG) experiments.
- Codes evaluated: (6, 4, 5), (12, 9, 11) and (20, 16, 19).

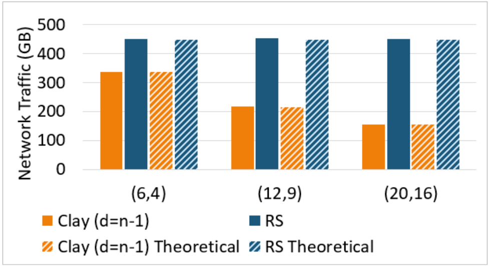
Network Traffic and Disk Read : W1 Workload, 1 PG



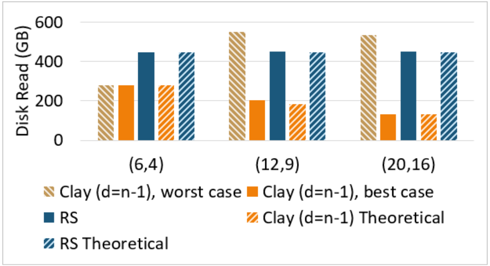


- Network traffic reduced to 75%, 48%, 34% of that of RS as predicted by theory.
- Repair disk read reduced to 62%, 41%, 29% of that of RS as predicted by theory.

Network Traffic and Disk Read : W2 Workload, 1 PG



- Network traffic reduced to 75%, 48%, 34% of that of RS matching the theoretical values.
- Reductions same as that for W1.



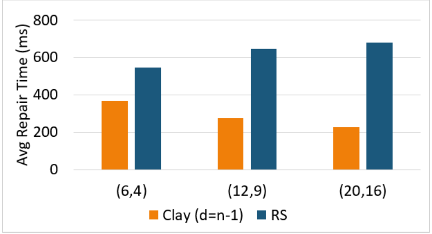
- Disk read for (6, 4, 5) code is optimal
- For (12, 9, 11) and (20, 16, 19) codes effect of fragmented read is observed.

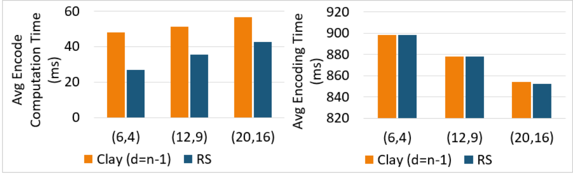
Fragmented Read

Best and worst case, disk read during repair of (20,16,19) code for stripe sizes 1MB, 64MB

- During repair of a chunk only $\beta < \alpha$ sub-chunks are read from each helper nodes.
- During worst case failures, the sub-chunks needed in repair are not located contiguously.
- sub-chunk size = stripe size$/k\alpha$
- For (20,16,19) code $\alpha = 1024$, $k = 16$. Therefore, for stripe sizes 64MB and 1MB, the sub-chunk sizes are 4KB, 64B respectively.
- If sub-chunk size is aligned to 4kB (SSD page granularity), the fragmented-read problem can be avoided.

Repair Time and Encoding Time: W1 Workload, 1 PG

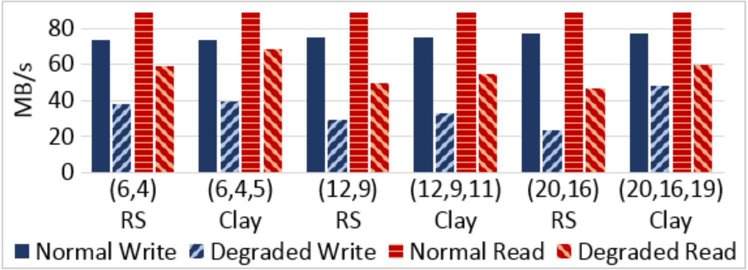




- Repair time reduced by 1.49x, 2.34x, 3x of that of RS.

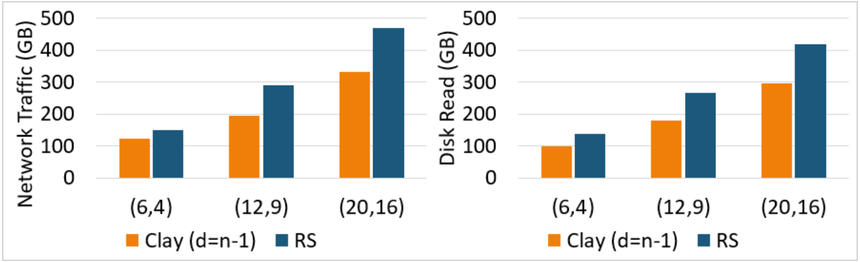
- The total encoding time remains almost same as that of RS.
- While, encode computation time of Clay code is higher than that of RS code by 70%.
- This is due to the additional PFT and PRT operations.

Normal and Degraded I/O : W1 workload, 1 PG



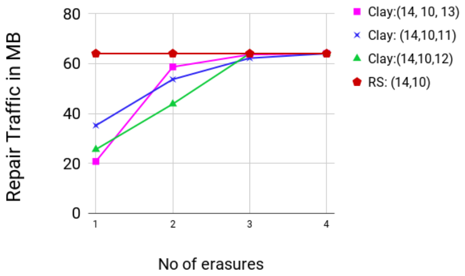
- Better degraded read 16.24%, 9.9%, 27.17% and write throughput increased by 4.52%, 13.58%, 106.68% of that of RS.
- Normal read and write throughput same as that of RS.

Network Traffic and Disk Read : W1 workload, 512 PG

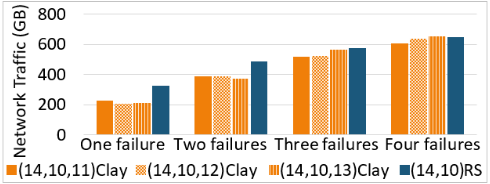


- Assignment of OSDs and objects to PGs is dynamic.
  - Number of objects affected by failure of an OSD can vary across different runs of multiple-PG experiment.
- Sometimes an OSD that is already part of the PG can get reassigned as replacement for the failed OSD.
  - Number of failures are treated as two resulting in inferior network-traffic performance in multiple-PG setting.

Multiple Node Failures



Average theoretical network traffic during repair of 64MB object.



- Workload W1, 512 PG
- Network traffic increases with increase in number of failed chunks.

Conclusions

- We provide an open-source implementation of Clay code for any (n, k, d) parameters.

Conclusions

- We provide an open-source implementation of Clay code for any (n, k, d) parameters.
- The theoretical promise of the Clay code is reflected in the evaluation presented here

Conclusions

- We provide an open-source implementation of Clay code for any (n, k, d) parameters.
- The theoretical promise of the Clay code is reflected in the evaluation presented here
- Specifically, for Workloads with large sized objects, the Clay code $(20, 16, 19)$:
  - resulted in repair time reduction by $3x$.
  - Improved degraded read and write performance by 27.17% and 106.68% respectively.

Conclusions

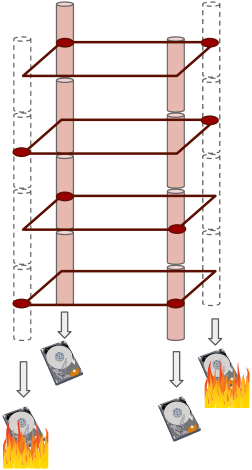
- We provide an open-source implementation of Clay code for any (n, k, d) parameters.
- The theoretical promise of the Clay code is reflected in the evaluation presented here
- Specifically, for Workloads with large sized objects, the Clay code $(20, 16, 19)$:
  - resulted in repair time reduction by $3x$.
  - Improved degraded read and write performance by 27.17% and 106.68% respectively.

In summary, Clay Codes are well poised to make the leap from theory to practice!!!

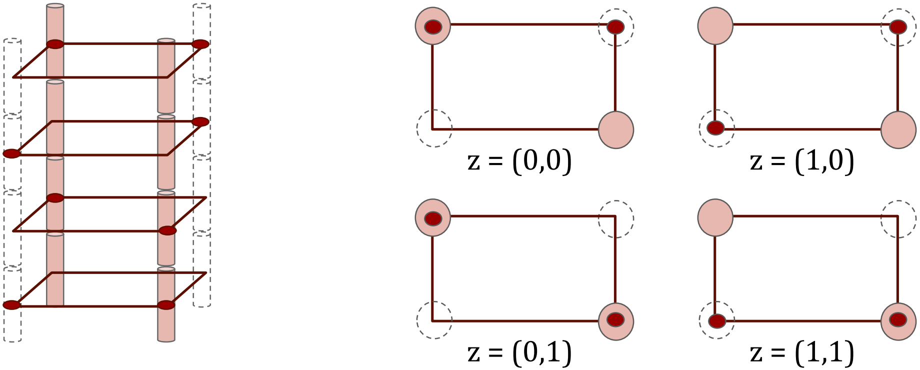
Thank You!

Backup Slides!

Decode: Two nodes fail

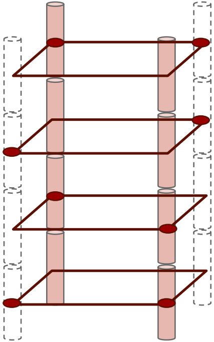


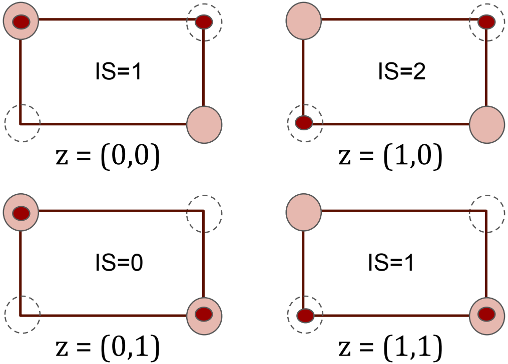
Assign Intersection Score to each plane



Intersection score is given by the number of hole-dot pairs

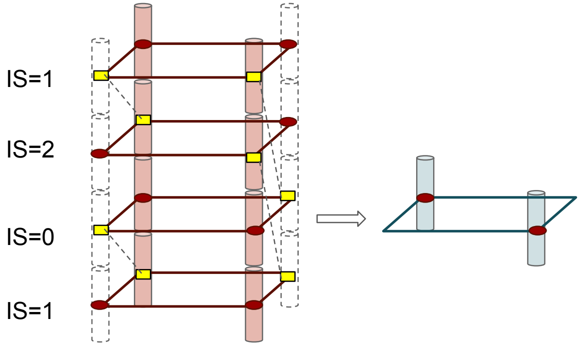
Assign Intersection Score to each plane



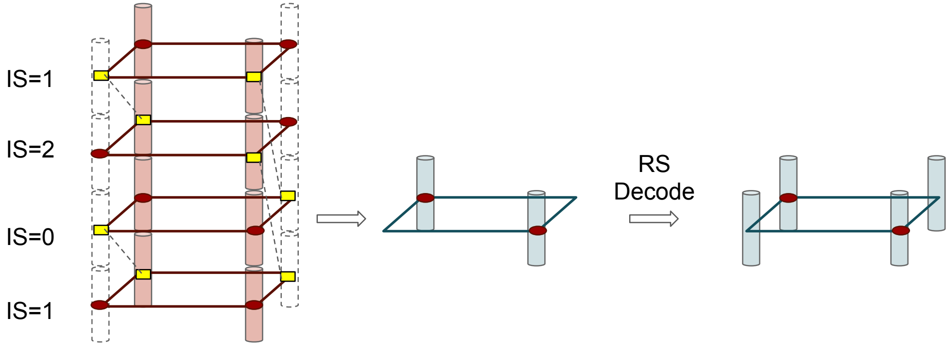


Intersection score is given by the number of hole-dot pairs

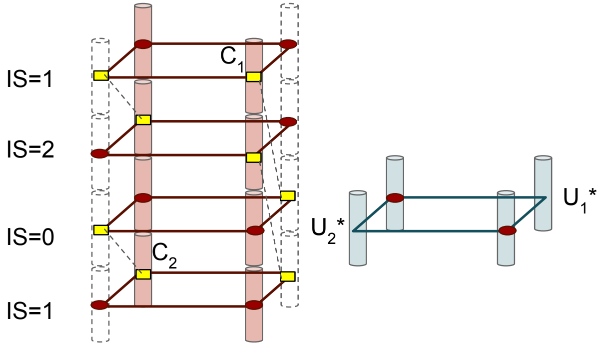
For non erased nodes, get the uncoupled sub-chunks for planes with IS=0



RS decode to get the remaining uncoupled-subchunks

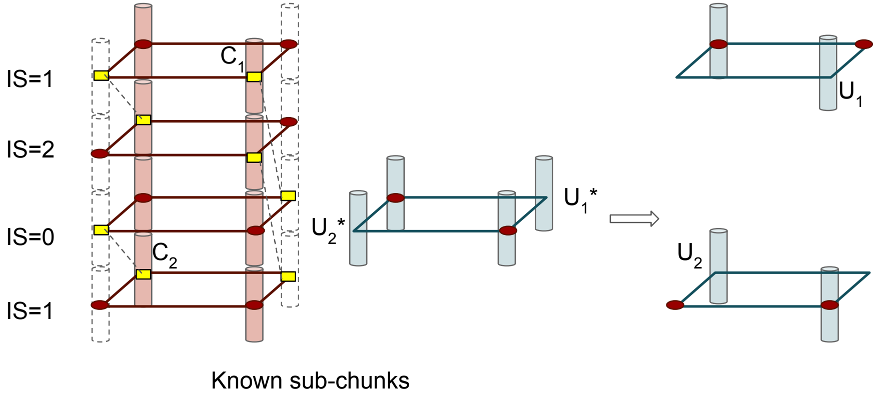


We now have following sub-chunks



Known sub-chunks

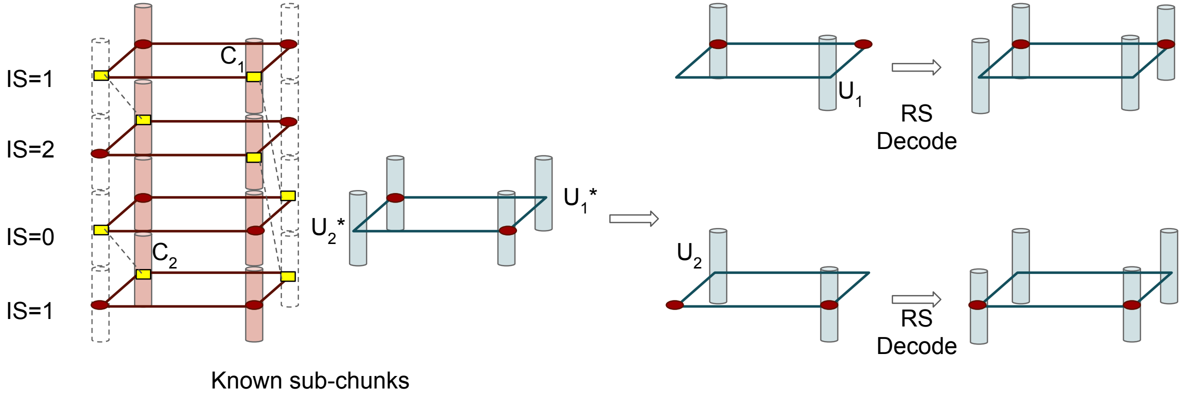
For non erased nodes, get the uncoupled sub-chunks for planes with IS=1



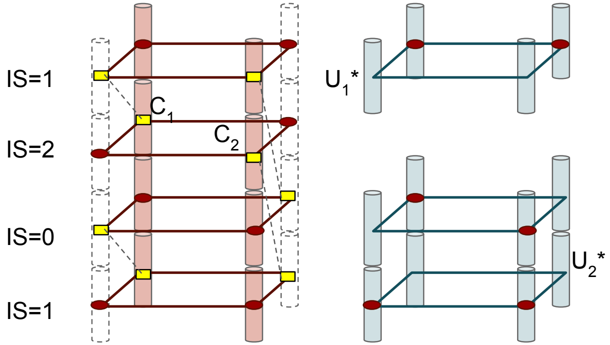
Get U_2 from U_2* and C_2

Get U_1 from U_1* and C_1

RS decode to get the remaining uncoupled-subchunks

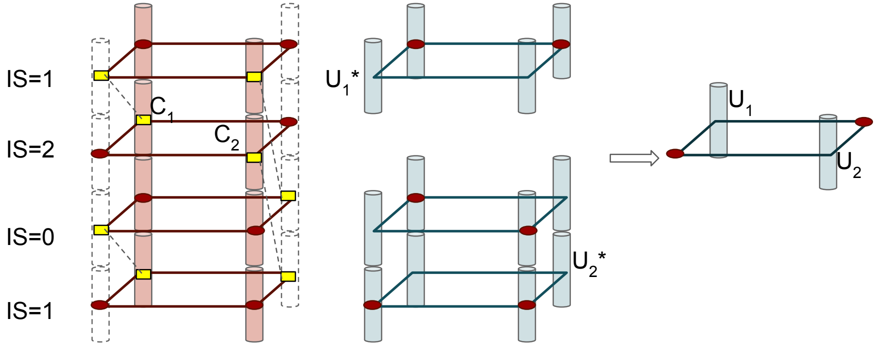


We now have the following sub-chunks



Known sub-chunks

For non erased nodes, get the uncoupled sub-chunks for planes with IS=2

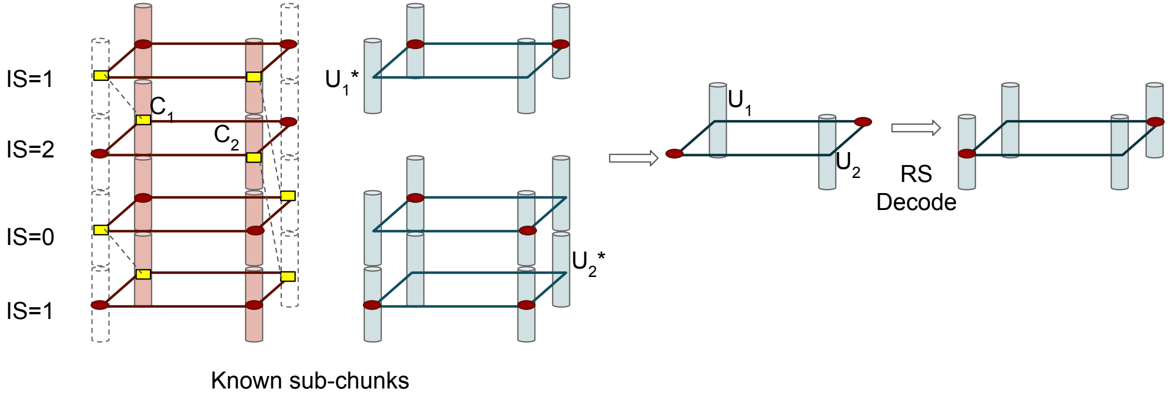


Known sub-chunks

Get U_2 from U_2* and C_2

Get U_1 from U_1* and C_1

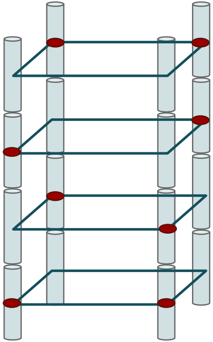
Get the uncoupled sub-chunks for planes with IS=2



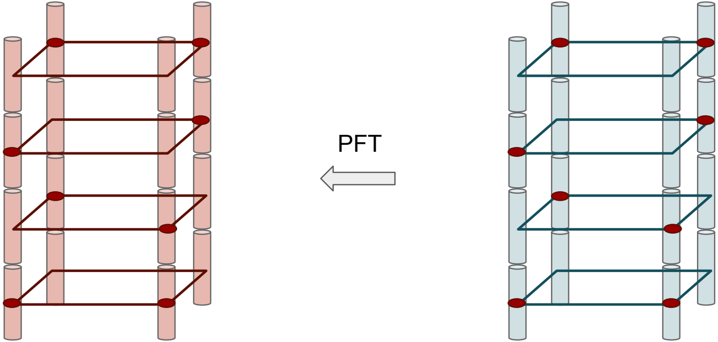
Get U_2 from U_2* and C_2

Get U_1 from U_1* and C_1

We now have all the uncoupled sub chunks



The coupled sub chunks can now be computed using PFT



The decoding is now complete