Codes for Big Data: Erasure Coding for Distributed Storage

P. Vijay Kumar

Professor,
Department of Electrical Communication Engineering
Indian Institute of Science, Bangalore

The 3rd Annual Storage Developer Conference
Bengaluru

May 25-26, 2017
Thanks go out to

- Paul Talbut and Udayan Singh for the invite

and

- K. Gopinath and Siddhartha Nandi

for being kind enough to suggest my name..
Acknowledgements

Research Collaborators Joint work with:

- Birenjith Sasidharan, Myna Vajha, S. B. Balaji and Nikhil Krishnan (PhD students, IISc)

- Bhagyashree Puranik, Ganesh Kini and Vinayak Ramkumar (MTech students, IISc)

- Srinivasan Narayanamurthy, Syed Hussain and Siddhartha Nandi (NetApp ATG, Bengaluru, India)
Organization

- Erasure Coding
- Node Failures and the Evolution of Coding Theory
- Regenerating Codes
- Locally Recoverable Codes (briefly)
- Codes with Local Regeneration (briefly)
- Codes for Multiple Erasures (briefly)
  - Codes for Data Availability
  - Codes with Sequential Recovery
- The Coupled-Layer MSR Code in Action
Erasure Coding
Fault Tolerance

- Fault tolerance is key to making data loss a very remote possibility.
- A time-honored means of achieving fault tolerance is replication.
Triple Replication

File or Data Object

Data Block

Triplet replication

Stored in different nodes of the storage network
Drawback of Triple Replication

- But triple replication is poor in terms of storage efficiency: just 33%. Are there better ways?
Drawback of Triple Replication

- But triple replication is poor in terms of storage efficiency: just 33%. Are there better ways?

- A well-known alternative is to use Erasure Coding (EC)
Erasure Coding of Data

File or Data Object

Split the data object into \( k \) parts

\( A_1 \quad A_2 \quad \ldots \quad A_k \)

\( k \) storage units

\( P_1 \quad P_2 \quad \ldots \quad P_m \)

add \( m \) parity storage units

\((k,m)\) erasure code
Two Key Performance Measures

1. Storage efficiency

\[ \frac{k}{k + m} \]

2. Fault tolerance

- at most \( m \) storage units

3. Codes with maximum possible fault tolerance \( \Rightarrow \) MDS codes

4. Reed-Solomon codes - a prime example
An Example MDS Code - The RAID 6 Code

Source: https://upload.wikimedia.org/wikipedia/commons/thumb/7/70/RAID_6.svg/1280px-RAID_6.svg.png
Other RS Codes in Practice

<table>
<thead>
<tr>
<th>Storage Systems</th>
<th>Reed-Solomon codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux RAID-6</td>
<td>RS(10,8)</td>
</tr>
<tr>
<td>Google File System II (Colossus)</td>
<td>RS(9,6)</td>
</tr>
<tr>
<td>Quantcast File System</td>
<td>RS(9,6)</td>
</tr>
<tr>
<td>Intel &amp; Cloudera’ HDFS-EC</td>
<td>RS(9,6)</td>
</tr>
<tr>
<td>Yahoo Cloud Object Store</td>
<td>RS(11,8)</td>
</tr>
<tr>
<td>Backblaze’s online backup</td>
<td>RS(20,17)</td>
</tr>
<tr>
<td>Facebook’s f4 BLOB storage system</td>
<td>RS(14,10)</td>
</tr>
<tr>
<td>Baidu’s Atlas Cloud Storage</td>
<td>RS(12,8)</td>
</tr>
</tbody>
</table>

Evolution of HDFS to Incorporate EC ⇒ HDFS-EC

1. Typically, EC reduces the storage cost by 50% compared with 3x replication.
2. Motivated by this, Cloudera and Intel initiated the HDFS-EC project.
3. Targeted for release in Hadoop 3.0.
4. Employs a striped layout:

5. Possibility of incorporating more sophisticated EC schemes!

---

Node Failures and the Evolution of Coding Theory
Node Failures

An important consideration is how efficiently the EC can handle node failures as such failures are commonplace:

Figure 1: Number of failed nodes over a single month period in a 3000 node production cluster of Facebook.

RS Codes and Node Failures

Under the conventional approach, RS codes are inefficient in two respects at node repair:

1. In the example Facebook [10, 4] RS code, the amount of data download (repair BW) equals 10 times the amount stored within the failed node.
2. Also, 10 storage units need to be contacted for repair.

There is room for improvement...
Coding Theory Responds

1. **Regenerating codes**
   - minimize the amount of data download (repair bandwidth) needed for node repair

2. **Locally recoverable codes**
   - minimize the number of helper nodes contacted for node repair, but also reduce repair bandwidth

3. **Novel and efficient approaches to RS repair** a more recent development

---

Some Comments

Regenerating Codes

1. Minimum Storage Regenerating (MSR) Codes are MDS codes
2. Regenerating codes are vector codes, each code symbol is a vector of code symbols
   ▶ $\ell$ is called the sub-packetization level

Locally Recoverable Codes

1. Locally recoverable codes yield on storage efficiency for ease of node repair

Fresh approach to RS repair

1. regard RS codes as vector codes
2. minimize repair bandwidth under a constraint on sub-packetization level $\ell$
Regenerating Codes

- Focus here on the subclass of Minimum Storage Regenerating (MSR) Codes
Raid Code - Not Very Good at Handling Node Failure.

The conventional approach:
- Connect to any 2 nodes,
- Reconstruct \( A \) and \( B \),
- Extract \( A \)

But downloading 2 units of data to revive a node that stores 1 unit of data is clearly, wasteful of network bandwidth..
Replacing the RAID 6 Code with a Regenerating Code

- Here, each node now stores two “half-symbols”
- We download 3 half-symbols as opposed to 2 full-symbols
  - Can recover any of \( \{A_1, A_2, B_1\} \)
## Evolution of MSR Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explicit</th>
<th>SE</th>
<th>SPL</th>
<th>OA</th>
<th>HN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product-Matrix</td>
<td>Yes</td>
<td>Low</td>
<td>Low</td>
<td>No</td>
<td>$d$</td>
</tr>
<tr>
<td>Hadamard &amp; Butterfly*</td>
<td>Yes</td>
<td>High</td>
<td>High</td>
<td>No</td>
<td>all</td>
</tr>
<tr>
<td>Zig-Zag Code</td>
<td>No</td>
<td>High</td>
<td>High</td>
<td>Yes</td>
<td>all</td>
</tr>
<tr>
<td>Sasidharan et al (1)</td>
<td>No</td>
<td>High</td>
<td>Low</td>
<td>Yes</td>
<td>all</td>
</tr>
<tr>
<td>Ye-Barg (1)</td>
<td>Yes</td>
<td>High</td>
<td>High</td>
<td>Yes</td>
<td>all</td>
</tr>
<tr>
<td>Ye-Barg (2)</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
<td>Yes</td>
<td>all</td>
</tr>
<tr>
<td>Sasidharan et al (2)</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
<td>No</td>
<td>$d$</td>
</tr>
</tbody>
</table>

* ⇒ limited to 2 parity nodes

- SE ⇒ storage efficiency
- SPL ⇒ sub-packetization level
- OA ⇒ optimal access (number of symbols accessed for repair)
- HN ⇒ number of helper nodes needed
References (MSR Codes with High Storage Efficiency)


Example Coupled-Layer MSR Code

- Our coupled-layer perspective on the Ye-Barg construction (2)
- a (4, 2) MSR code
- 6 nodes, sub-packetization level is \( \ell = 8 \)
- \( 6 \times 8 = 48 \) points
- in the example to follow, each point stores 2MB

---

2. B. Sasidharan, M. Vajha, and PVK. “An Explicit, Coupled-Layer Construction of a High-Rate MSR Code with Low Sub-Packetization Level, Small Field Size and \( d < (n - 1) \),” to be presented at ISIT 2017.
A comparison of actual repair time is shown. In the figure,
- the (6, 4) code is in our present notation a (4, 2) code
- the (12, 9) code is in our present notation a (9, 3) code
- the (20, 16) code is in our present notation a (16, 4) code
Performance of the Coupled-Layer MSR Code

- Similar gains in network bandwidth and disk read

- Thus a larger sub-packetization level is not necessarily a problem for implementation
Locally Recoverable Codes
Comparison: In terms of reliability and number of helper nodes contacted for node repair, the two codes are comparable. The overheads however are quite different, 1.29 for the Azure code versus 1.5 for the RS code. This difference has reportedly saved Microsoft millions of dollars.

Codes with Hierarchical Locality

- $[4, 3, 2]$ code $\Rightarrow$ (3,1) code
- $[12, 8, 3]$ code $\Rightarrow$ (8,4) code
- $[24, 14, 6]$ code $\Rightarrow$ (14,10) code

Codes with hierarchical locality do exactly that by calling for help from an intermediate layer of codes when the local code fails.

These codes may be regarded as the “middle codes”.

---

Codes with Local Regeneration
Codes with Local Regeneration

- A single code that has both locality and regeneration properties
- and inherent double replication of data

An Example Code with Local Regeneration

The construction makes can make use of an all-symbol local scalar code and is also optimal:

Local Code 1

Local Code 2

Local Code 3

Scalar All-Symbol Locality Code

Local Code 1

Local Code 2

Local Code 3
Codes with Availability (Recovery from Simultaneous Multiple Erasures)
Recovery in Parallel

<table>
<thead>
<tr>
<th></th>
<th>$c_{11}$</th>
<th>$c_{12}$</th>
<th>$c_{13}$</th>
<th>$c_{14}$</th>
<th>$c_{15}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{21}$</td>
<td>$c_{22}$</td>
<td>$c_{23}$</td>
<td>$c_{24}$</td>
<td>$c_{25}$</td>
<td></td>
</tr>
<tr>
<td>$c_{31}$</td>
<td>$c_{32}$</td>
<td>$c_{33}$</td>
<td>$c_{34}$</td>
<td>$c_{35}$</td>
<td></td>
</tr>
<tr>
<td>$c_{41}$</td>
<td>$c_{42}$</td>
<td>$c_{43}$</td>
<td>$c_{44}$</td>
<td>$c_{45}$</td>
<td></td>
</tr>
<tr>
<td>$c_{51}$</td>
<td>$c_{52}$</td>
<td>$c_{53}$</td>
<td>$c_{54}$</td>
<td>$c_{55}$</td>
<td></td>
</tr>
</tbody>
</table>

- Last column is a parity check on entries to the left in the same row
- Last row is a parity check on entries above in the same column
- Can recover *locally* from 2 erasures in parallel
Codes with Sequential Recovery (Recovery from Simultaneous Multiple Erasures)
Sequential Recovery

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{11}$</td>
<td>$c_{12}$</td>
<td>$c_{13}$</td>
<td>$c_{14}$</td>
<td>$c_{15}$</td>
</tr>
<tr>
<td>$c_{21}$</td>
<td>$X$</td>
<td>$c_{23}$</td>
<td>$c_{24}$</td>
<td>$c_{25}$</td>
</tr>
<tr>
<td>$c_{31}$</td>
<td>$X$</td>
<td>$X$</td>
<td>$c_{34}$</td>
<td>$c_{35}$</td>
</tr>
<tr>
<td>$c_{41}$</td>
<td>$c_{42}$</td>
<td>$c_{43}$</td>
<td>$c_{44}$</td>
<td>$c_{45}$</td>
</tr>
<tr>
<td>$c_{51}$</td>
<td>$c_{52}$</td>
<td>$c_{53}$</td>
<td>$c_{54}$</td>
<td>$c_{55}$</td>
</tr>
</tbody>
</table>

- Same code as before
- Can recover locally from 3 erasures *in a sequential manner*
- Sequential recovery enables codes with larger storage efficiency
References - Codes for Multiple Erasures


Functioning of an Example, Coupled-Layer MSR Code

- Goal: To show that a larger sub-packetization level is not necessarily a problem for implementation
Example Coupled-Layer MSR Code

- Our coupled-layer perspective on the Ye-Barg construction (2)
- a (4, 2) MSR code
- 6 nodes, sub-packetization level is $\ell = 8$
- $6 \times 8 = 48$ points
- in the example to follow, each point stores 2MB

Consider a file of size 64MB

• Will encode via a $[k=4, m=2]$ MSR Code
• Called the Coupled-Layer MSR Code
Step 1: Break file into $k = 4$ data chunks, each of 16MB.
Data cube representation of CL-MSR Code

The cube has:

- 6 columns, each associated to a distinct node
- 8 horizontal planes.
- A column has 8 points
- Each point corresponds to 2MB of storage
Place four 16MB chunks in four systematic nodes.
Place four 16MB chunks in four systematic nodes
Place four 16MB chunks in four systematic nodes
Place four 16MB chunks in four systematic nodes
We now have the systematic nodes
We will now compute the parity nodes
Will get there through an intermediate “Virtual data cube”
Start filling the virtual data cube on the right as follows
Certain pairs of points in the cube are “coupled”
The Coupling Transform is a 2x2 matrix transform
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Place the points obtained in the Virtual data cube
Red dotted points are not paired, they are simply carried over.
Red dotted points are not paired, they are simply carried over
We now have data-part of the Virtual data cube
Each plane is Reed-Solomon coded to obtain parity points.

\[ Z = (0,0,0) \]
Each plane is Reed-Solomon coded to obtain parity points

\[ z = (0,0,0) \]
Each plane is Reed-Solomon coded to obtain parity points
Each plane is Reed-Solomon coded to obtain parity points.
Each plane is Reed-Solomon coded to obtain parity points
Each plane is Reed-Solomon coded to obtain parity points.
Each plane is Reed-Solomon coded to obtain parity points.
Each plane is Reed-Solomon coded to obtain parity points.
Each plane is Reed-Solomon coded to obtain parity points
Each plane is Reed-Solomon coded to obtain parity points

$Z = (0,1,1)$

RS Encode
Each plane is Reed-Solomon coded to obtain parity points.
Now we have the complete Virtual data cube
Parity points of Actual data cube can now be computed
Perform decoupling

Virtual data cube

B
Perform decoupling

Virtual data cube
B
Perform decoupling

Virtual data cube B
Perform decoupling

Virtual data cube B

A\textsubscript{1} A\textsubscript{2}
Perform decoupling
Perform decoupling

Virtual data cube

Inverse Coupling Transform

B₁ B₂

A₁ A₂
Perform decoupling

Virtual data cube

B

A1  A2
Perform decoupling
Perform decoupling
Perform decoupling

Virtual data cube B
Perform decoupling

Virtual data cube B
Perform decoupling

Virtual data cube B
Red dotted points are simply carried over.
Red dotted points are simply carried over
Decoupling

Coupling

Virtual data cube

A

Decoupling

Coupling

Virtual data cube

B
The encoding is now completed!
Problem of Node Repair: One node fails
Problem of Node Repair: One node fails
For this example, only half of the planes participate in repair

- Total Helper Data = 2MB × 4 × 5 = 40MB
- Opposed to RS code = 16MB × 4 = 64MB
- Much larger savings seen for m > 2
Coupling points
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
Half the number of required points are now already computed
Remaining points are computed by coupling transform
Remaining points are computed by coupling transform
Remaining points are computed by coupling transform
Remaining points are computed by coupling transform
Contents of the failed node are now completely recovered.
Node Repair done: system back to original state!
Thanks!