# High Parformance Computing

2015/1/26

Naohiro Ohta

Wakita Lab

# Today's Paper

Title: CORE:Cross-Object Redundancy for Efficient Data Repair in Storage Systems

Authors: Kyumars Sheykh Esmaili
Lluis Pamies-Juarez
Anwitaman Datta

2013 IEEE International Conference on BigData

# Abstraction

- Fault tolerance for Big Data
- erasure code
- Cross−object Redundancy

# Back ground -fault tolerance for data

- This paper deals with how to repair the fault with large amount of data

- Traditional approach is erasure code

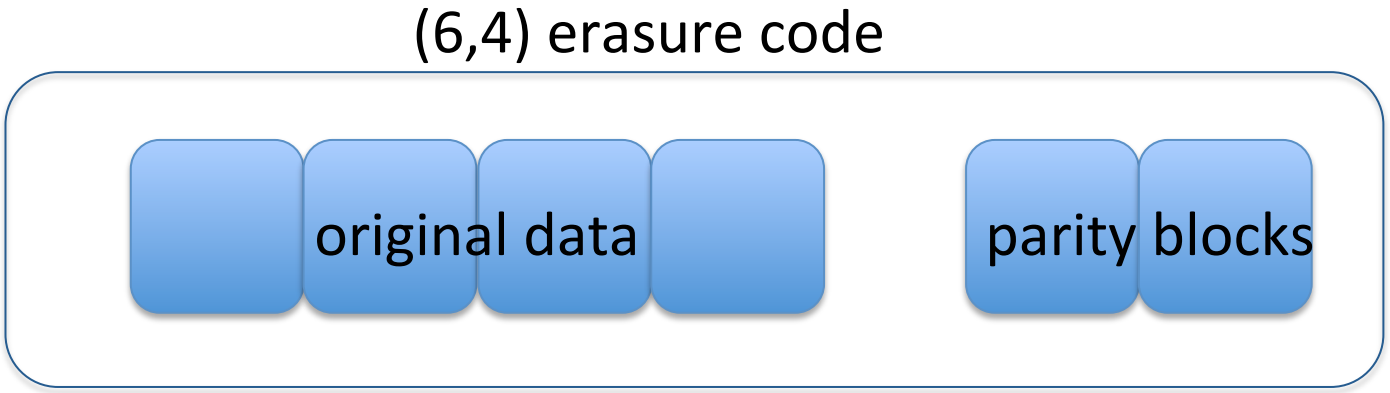
# Background- Classic erasure codes

- (n,k) erasure code

  split data into k blocks of size q

  computes m = n-k parity blocks  and store

  different node



# Classic Erasure Codes

- data vector $o = (o_1,..,o_k)$

  Generater matrix G and n-dimentional codeword

  $c = (c_1,..,c_n) = [o,p] = o \cdot G$

  P is parity vector

  $G = [I_k,G']$ $G'$ = k × m matrix

# Classic Erasure Codes

- This process stretches the original data by n/k
- Trade−off between storage overhead and fault tolerance
- MDS has property o can reconstructed from any k out of total n stored blocks

# MDS(Addition)

- Singleton Bound

  $|C| \leqq a^{n-d+1}$

(|C| is number of codewords, n is a code length, d is minimum distance, a is number of alphabets )

if the left equals the right ,the code is called MDS (Maximum Distance Separable) code

- In this case, $|C| = 2^q$,a = 2, n = q, d = 1

# Background - Locally Repairable Codes

- Repairing a single failed block requires to download an amount of Information

- Reducing the number of blocks needed to carry out the repair or reconstruction

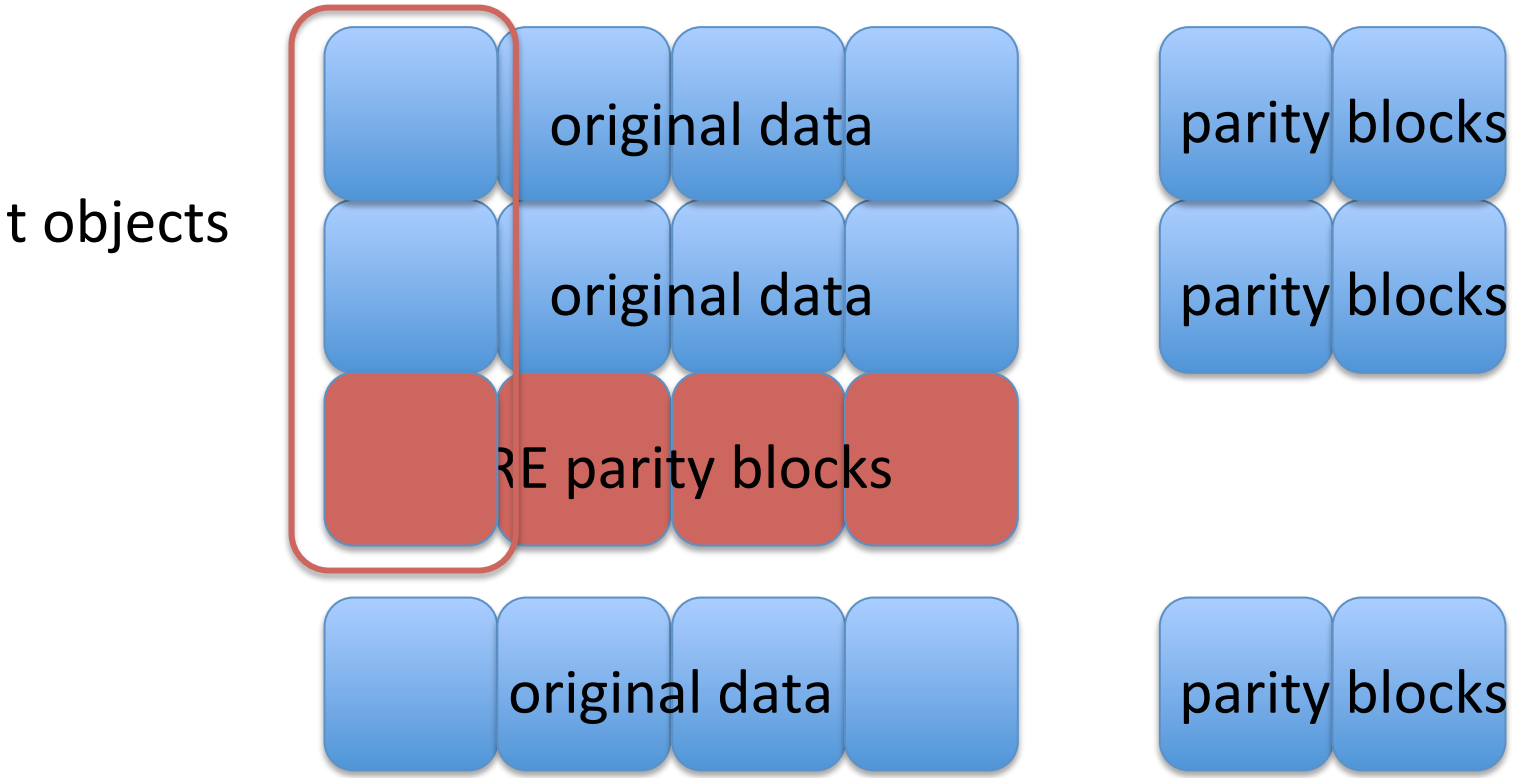
# Locally repairable codes

- If local block $c_i$ can be expressed

  $c_i = \alpha_1 c'_1 + \ldots + \alpha_d c'_d \qquad (c'_l /= c_i)$

- If d = 2 , the tolerance is poorer than MDS
- Trade off three properties
  - High fault tolerance
  - low storage overhead
  - efficient repairs and degraded reads

# Cross-object redundancy

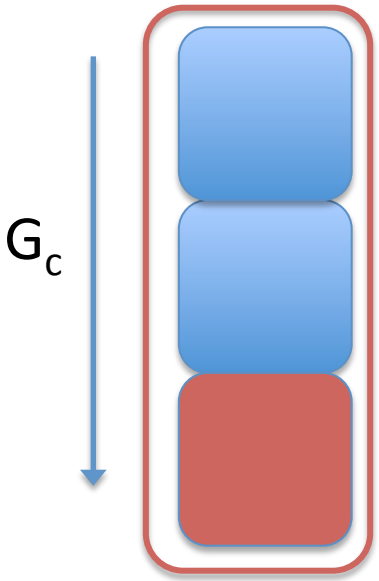



Making vertical parity codes

# CORE's product code definition

$G_c$ ,$G_o$ ‥ generator matrices of $(n_c,k_c)$ & $(n_o,k_o)$

$G = G_c \otimes G_o = [I_t,1_t] \otimes [I_k,H]$




$$H = \begin{pmatrix} \alpha_1^0 & \cdots & \alpha_1^{m-1} \\ \vdots & \ddots & \vdots \\ \alpha_k^0 & \cdots & \alpha_k^{m-1} \end{pmatrix},$$

$$\alpha_i \in F_{2^q}$$

# CORE'S product code definition

good reparability (small t)

good storage overhead(large t)

$t \approx k/2$

CORE`S product code (n,k,t) uses G

# CORE's algorithmic aspects

- CORE works with matrix of t objects
- Dealing with algorithmic aspects for implements
  - Identifying independent clusters
  - Recoverability-checking algorithms
  - Repair scheduling algorithms

# Identifying Independent Clusters

- Define disjoint subsets of failed nodes as independent clusters with single failure
  - 2 cluster must not share any common row or column containing failed nodes

- To repair parallel
- To recover partially when Full CORE matrix in not recoverable

# Identifying Independent Clusters

- Merging single failure clusters

  if there exists common row or column on which both clusters have a failure

- Doing this until no mergeable clusters left

# Identifying Independent Clusters

- 10M random generate
- Reduce numbers of clusters greater 6


Figure 3: The average number of clusters versus the number of failures for CORE's code parameters (14,12,5)

# Recoverablity –Checking Algorithm

For (n,k,t) code,

- Lower bound of irrecoverablity L

  $L = 2\times(n - k + 1)$

- Upper bound of recoverablity U

  $U = t \times (n - k) + (2k - n)\times l$

F < L : recoverable

L =< F < U : containing both cases

L < F : irrecoverable ∗ F: failre number

# Recoverablity –Checking Algorithm

- L = 6 , U = 20

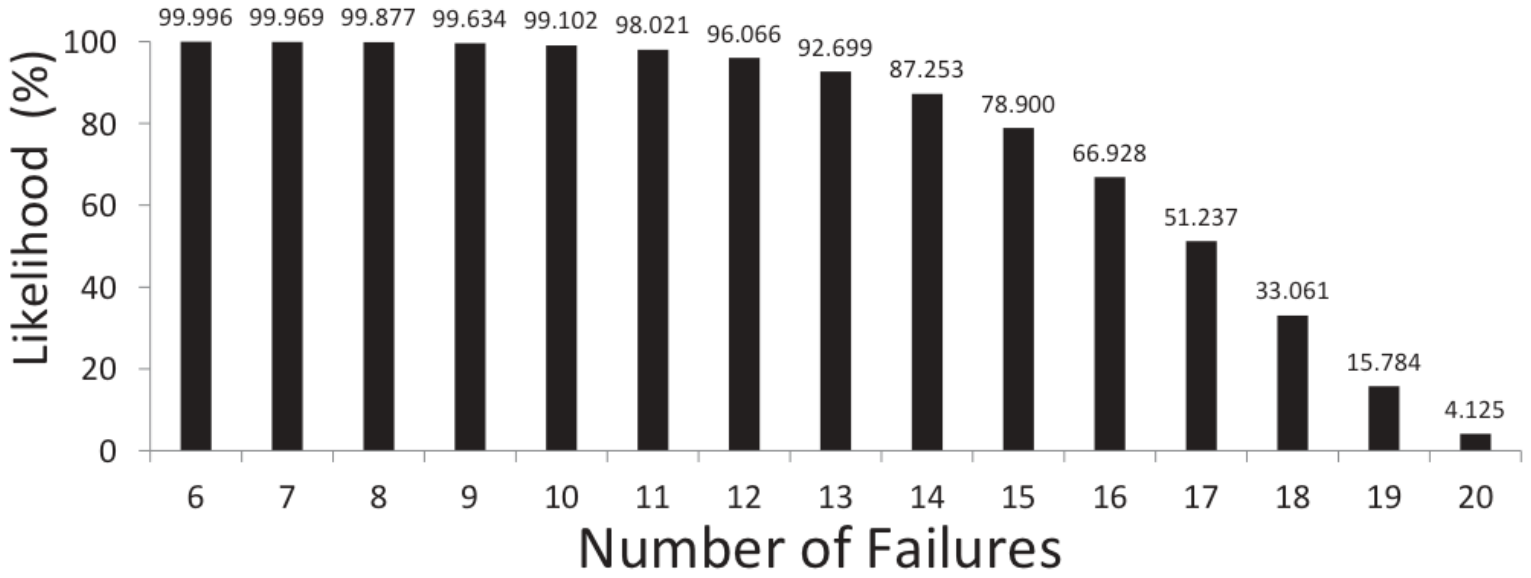



Figure 4: The recoverability likelihood of the scheme (14,12,5) based on the number of failures.

# Repair scheduling algorithm

- Column−first algorithm

higher priority to vertical repairs

- Row−first algorithm

higher priority to horizontal repairs

# Repair scheduling algorithm

- Recursively generated schedule algorithm (RGS)

$$v = \sum_{i=1}^{t} minV(Row_i) \qquad ; \qquad h = \sum_{j=1}^{k} minH(Col_j)$$

$$minV(Row_i) = \begin{cases} 0 & \text{if } |X| \leq (n-k) \\ |X| - (n-k) & \text{otherwise} \end{cases}$$

$$minH(Col_j) = \begin{cases} 0 & \text{if } |X| \leq 1 \\ |X| - 1 & \text{otherwise} \end{cases}$$

# Repair Scheduling algorithm

$$c(h,v) = \begin{cases} c(h, dec(v)) + t & \text{if } v > 0 \\ c(dec(h), v) + k & \text{if } v = 0 \\ & \text{or } dec(v) \text{ is not applicable} \end{cases}$$

- dec(v or h) reflect the decreases of v or h
- computing until c(0,0)

# Repair scheduling algorithms

- RGS and Column-first are better
- For Large failure numbers ,the difference of three algorithm is less

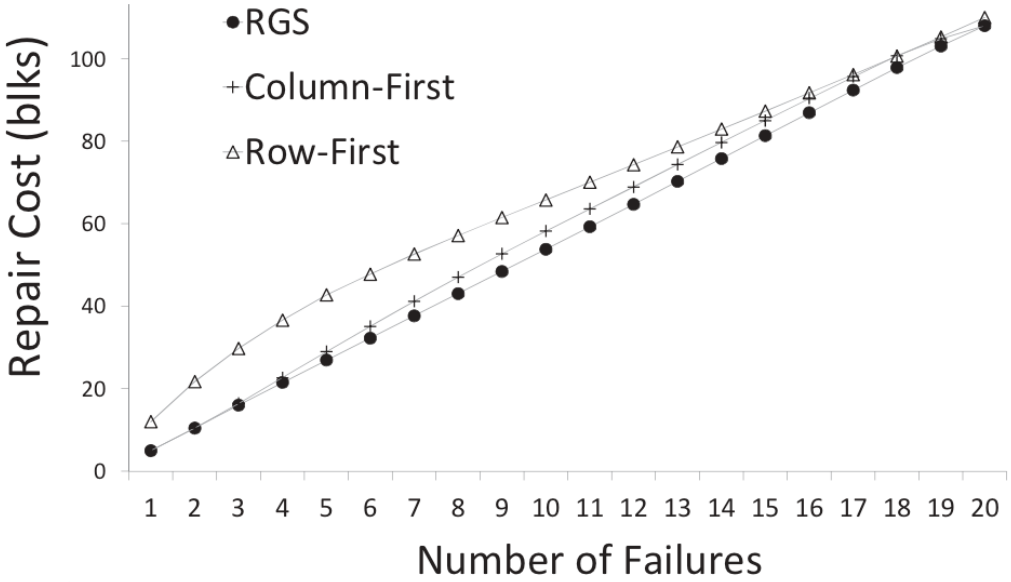



Figure 5: Comparing the Column-First, Row-First, and RGS algorithms w.r.t number of blocks required to carry out the repair on the scheme (14,12,5).

# Implementation

- They used HDFS−RAID
  - Open sorce module providing basic erasure code for Apache hadoop file system
- two Optimizations

- Implementation of CORE is used these

# HDFS–RAID

- Supporting both Reed-Solomon coding and XOR parity file
- RaidNode
  - a deamon responsible for the creation and maintenance of parity files
- BlockFixer
  - reconstructs missing or corrupt blocks

# HDFS-RAID Optimizations

- Opt1

HDFS uses all the remaining blocks to repair missing one, But They used exactly k blocks

- Opt2

Checking multiple failures per row(stripe)

# CORE Implementation

- RAIDing

  vertical Coding across files in a given directory

- Repair

  they implements all algorithm aspects

  – failure dictation and failure matrix population
  – clustering
  – recoverablity checking
  – repair scheduling

# Experiments

- Network−Critical cluster

  university cluster

  one PC and 19 Datanodes
- Computation-Critical cluster

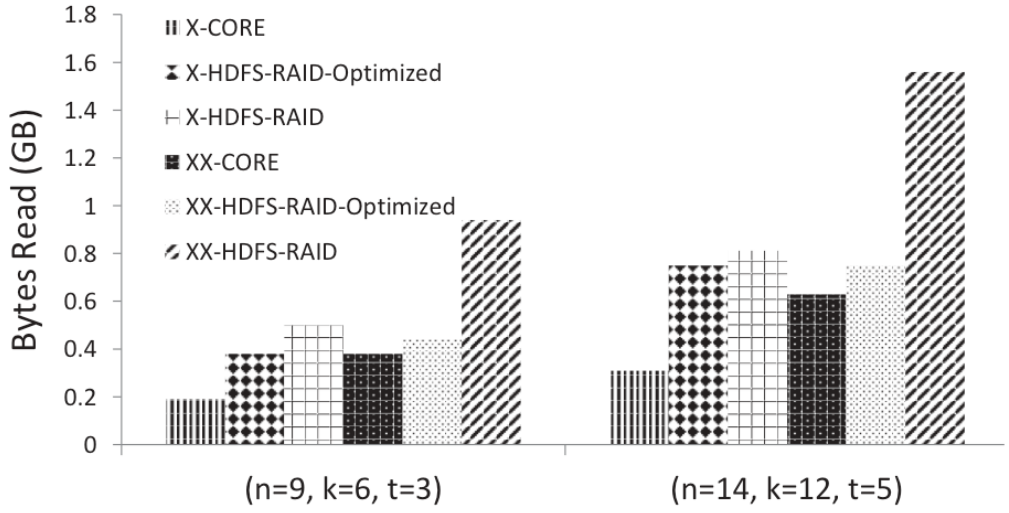
  20 node Amazon EC2 cluster

evaluation with completion time and transferred data

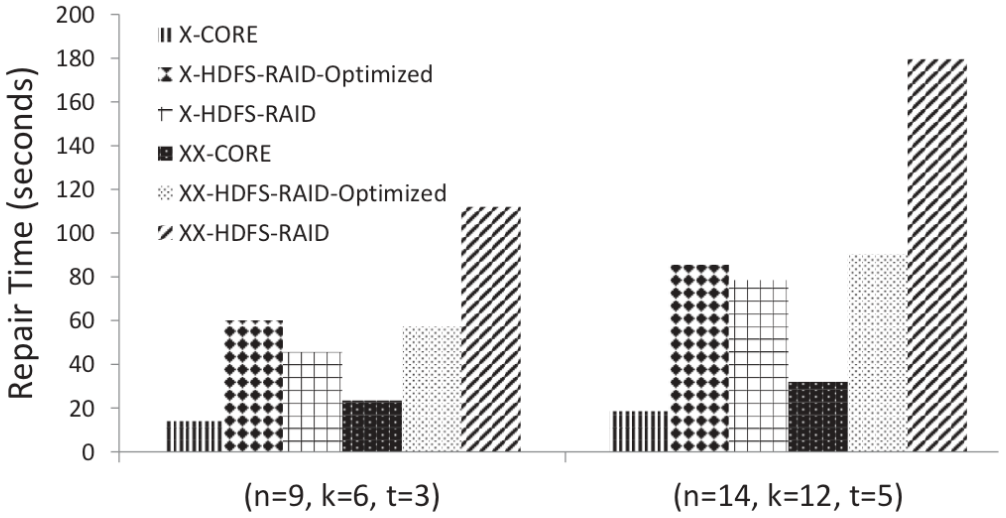
# Experiments

CORE VS HDFS-RAID

- X : one-failure
- XX: two-failure




(a) Transferred data

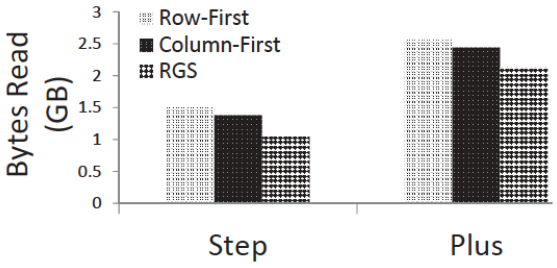



(c) Time (computation-critical cluster)

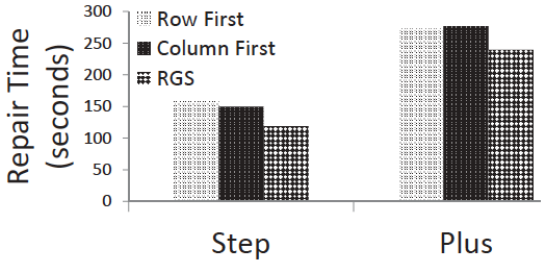
Figure 6: Comparing the repair performance of HDFS-RAID, HDFS-RAID-Optimized, and CORE

# Experiments

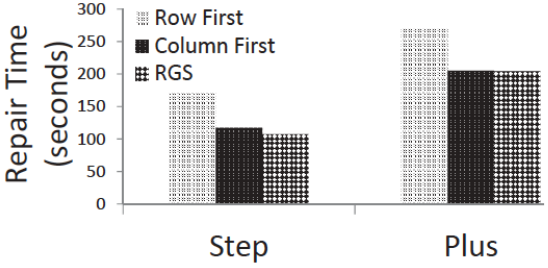
- Repair scheduling algorithms
  - CORE has the best results
  - Column-first is longer than expected in (b)
  - RGS is only slightly better in (C)




(a) Transferred data




(b) Time (network-critical cluster)




(c) Time (computation-critical cluster)

Figure 7: Performances of the repair scheduling algorithms on two different failure patterns.

# Conclusion and Future work

- Introducing cross object code
- Some algorithm about erasurre code
- Show  better performance than HDFS-RAID

- Better performance also during data insertion and updates
- They will carry out trace driven experiments to study the system dynamics better