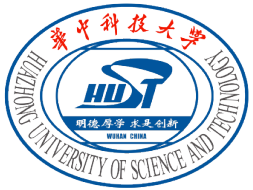
PivotRepair: Fast Pipelined Repair for Erasure-Coded Hot Storage

Qiaori Yao[1] , Yuchong Hu[1], Xinyuan Tu[1], Patrick P. C. Lee[2],
Dan Feng[1] , Xia Zhu[3], Xiaoyang Zhang[3], Zhen Yao[3], Wenjia Wei[3]

[1] *Huazhong University of Science and Technology*
[2] *The Chinese University of Hong Kong*
[3] *HUAWEI*

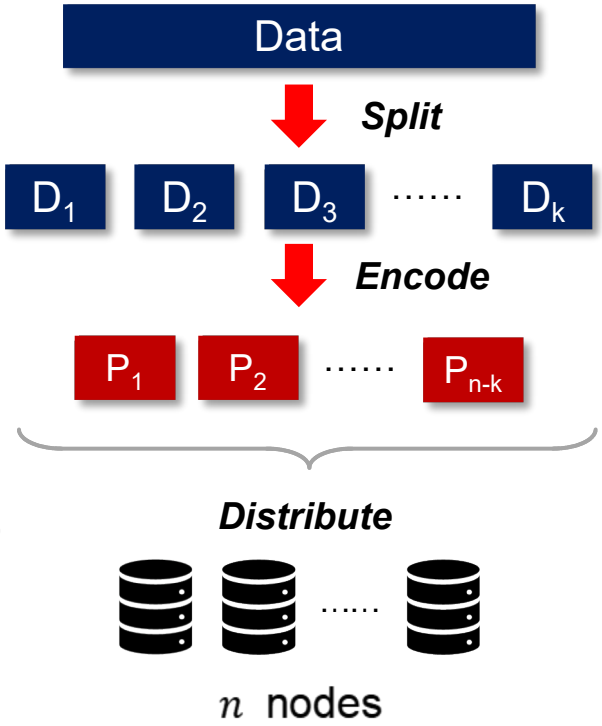
Massimo Gallo[3]



HUAWEI

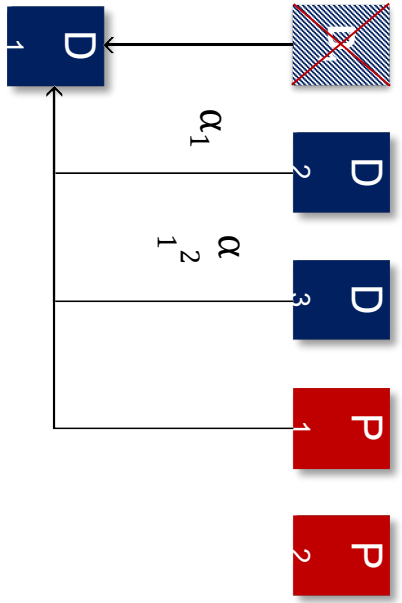
Erasure Coding

- A Popular Technique for Fault Tolerance
  - Protects data via **encoding.**
  - **Low degree of redundancy.**
  - An alternative to replication.
- Reed-Solomon (RS) Codes
  - Bases on Galois Field $GF(2^w)$.
  - (n, k): k data chunks => $n-k$ parity chunks.
  - Rebuilds a data chunk with k out of $n-1$ chunks.
  - Linearity:
    1. Additions keep the data size unchanged.
    2. Additions are associative.

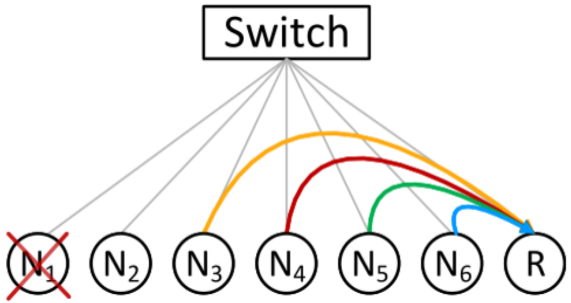


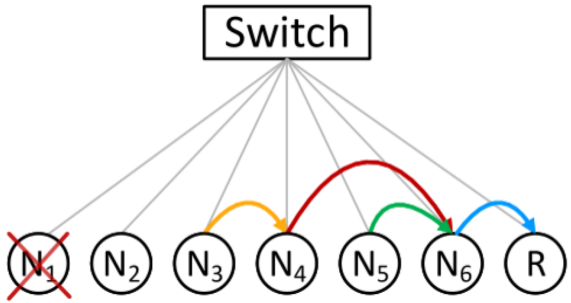
Repair in Erasure Coding

- Expensive Repair
  - Is triggered when reading unavailable data.
  - Single-chunk repair and full-node repair.
  - Needs to retrieve k chunks to repair a single chunk.
  - **A huge amount of data transferred for repair.**
- Studies on Repair
  - Reduces the repair traffic
    - Repair-friendly erasure codes (e.g., regenerating codes and locally repairable codes).
  - Distributes the repair traffic
    - Parallelizes and pipelines the repair (e.g., PPR, RP and PPT).



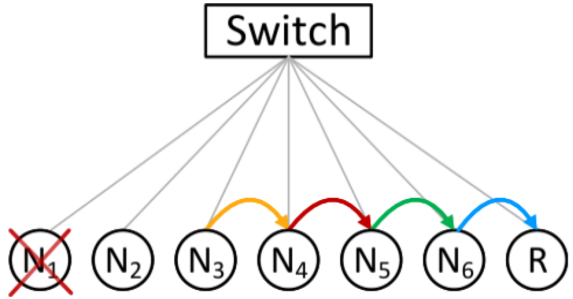
Current Repair Strategies

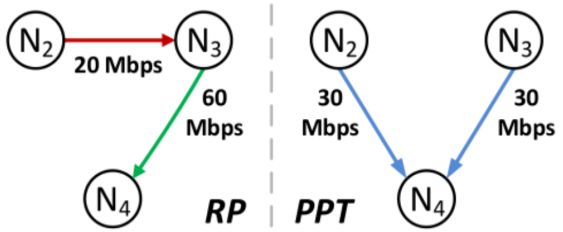




- Conventional Repair
  - The requestor collects all chunks to repair.
  - **The requestor is bottlenecked.**
  - Repair time: $O(k)$
- Partial-Parallel-Repair (PPR)
  - Decomposes a repair operation into **parallel partial sub-operations.**
  - Performs simultaneously.
  - Repair time: $O(logk)$

Current Repair Strategies





- Repair Pipelining (RP)
  - **Pipelines** the repair operation across helpers in **sub-chunks** (chain-like path).
  - Repair time: $O(1)$

- Parallel Pipeline Tree (PPT)
  - Addresses the repair in non-uniform traffic environments with a **tree-like structure**, to avoid slow links.
  - But its algorithm may incur an **exponential time complexity**.

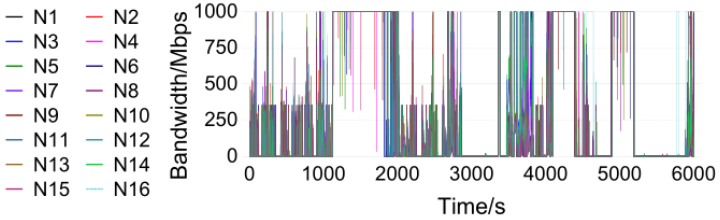
Erasure-Coded Hot Storage

- Be different from Cold Storage
  - Stores frequently accessed data.
  - Requires fast response time and low I/O latencies.
  - Requires **fast online recovery** to preserve read performance.
  - The network bandwidth is often shared by both repair and foreground jobs.
  - Frequent and rapidly-changing network congestions.

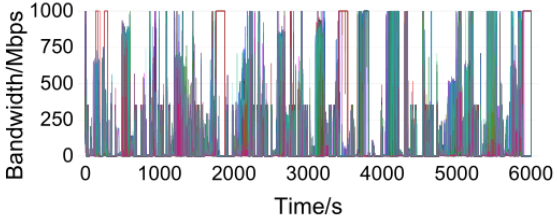
Can existing repair strategies handle the complex network environment in hot storage?

Observation

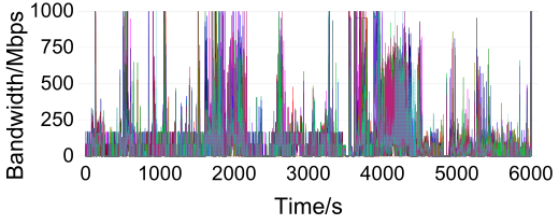
- Measurement analysis
  - Three hot storage workloads:
    1. TPC-DS, featuring the throughput of queries.
    2. TPC-H, featuring business databases.
    3. SWIM, a real MapReduce trace from Facebook.
  - Set up a Hadoop cluster of 16 machines at Amazon EC2.



(a) TPC-DS



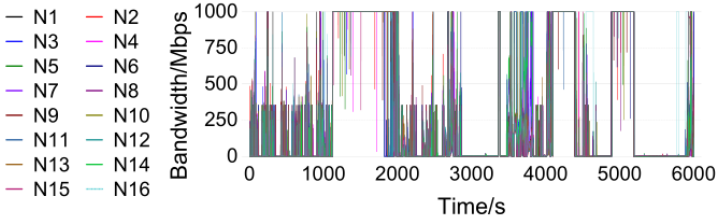
(b) TPC-H



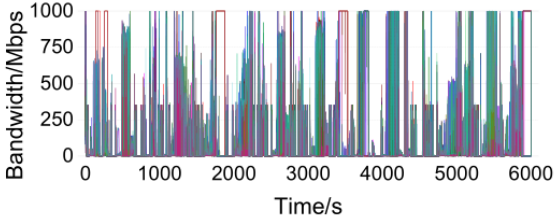
(c) SWIM

Observation

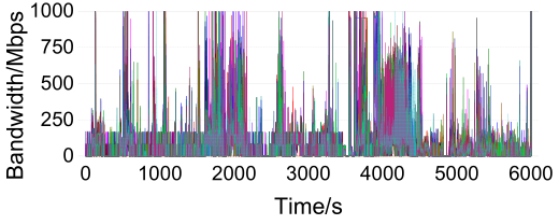
- Observation results
  1. **Rapidly-changing bandwidths** : frequent and rapidly-changing congested nodes caused by application jobs in hot storage may bottleneck the repair job.
  2. **Heterogeneous bandwidths**: there still exist some uncongested nodes with relatively sufficient available downlink and uplink bandwidths.



(a) TPC-DS



(b) TPC-H

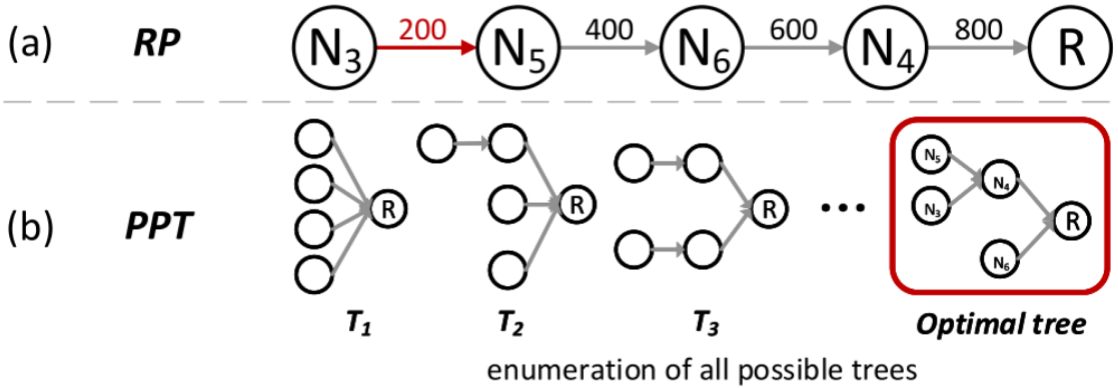


(c) SWIM

Motivation

- Drawbacks of Existing Strategies
  - RP's chain-like path cannot rapidly **bypass** network congestions.
  - PPT cannot solve for the most suitable pipelined tree **in a short time** (exponential time complexity).

	N_3	N_4	N_5	N_6	R
Available Uplink Bandwidth (Mbps)	600	800	510	600	1000
Available Downlink Bandwidth (Mbps)	130	1000	200	400	1000

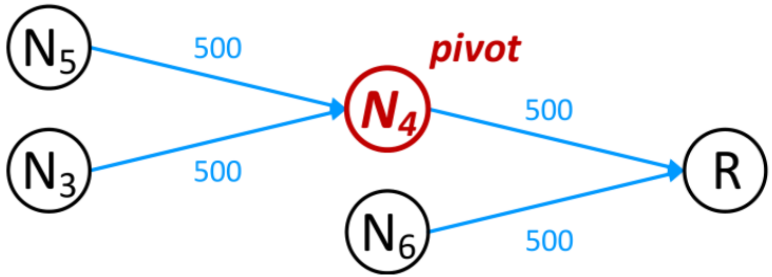


Motivation

- Our main idea
  1. Bypasses the congested nodes (outperforms RP) using the uncongested nodes to relay the repair traffic.
  2. Accelerates the pipelined tree construction (outperforms PPT) using the uncongested nodes to construct the tree in advance.

	N_3	N_4	N_5	N_6	R
Available Uplink Bandwidth (Mbps)	600	800	510	600	1000
Available Downlink Bandwidth (Mbps)	130	1000	200	400	1000

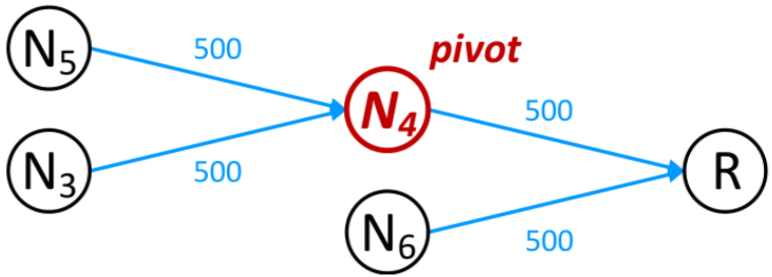
(c) ***Our Idea***



PivotRepair

- Exploiting Uncongested Nodes
  - **Pivots**: the uncongested nodes in a storage network.
  - Goal: constructs an optimal and fast-constructed pipelined repair tree.
  - Optimal: bypasses congested nodes, maximizes the bandwidth of the slowest stage in the pipelined repair tree.
  - Mechanism:
    - Uses pivot-based non-leaf nodes to relay the repair traffic to avoid congestion.
    - Uses pivot-based non-leaf nodes to determine parts of the pipelined tree quickly.

(c) ***Our Idea***



Algorithm Details

- Tree Construction
  - B_{min}: the bandwidth of the slowest link of the repair pipelined tree.
  - Overview: using pivots to construct the tree and maximize B_{min} in stages.
  - Procedure:
    - Preparation: sort to get k pivots and their inserting order.
    - **Step 1 (Inserting)**: inserts k pivots into the tree on by one to construct a preliminary tree, and try to maximize B_{min} greedily; reduces complexity via priority queue.
    - **Step 2 (Replacing)**: replaces some leaf nodes with unselected nodes to bypass congested leaf nodes, trying to further increase B_{min}.

Algorithm Example

- **Preparation**
- **Step 1 (Inserting)**

	N_2	N_3	N_4	N_5	N_6	R
Uplink (Mbps)	750	500	150	500	500	980
Downlink (Mbps)	100	130	1000	200	900	980

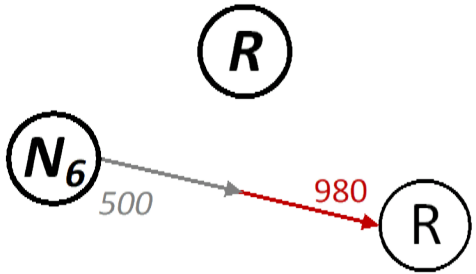
R is the requestor, sort other nodes in terms of ***theo*(·)**:

S : N_6, N_5, N_4, N_3, N_2

Insert R as the root node:

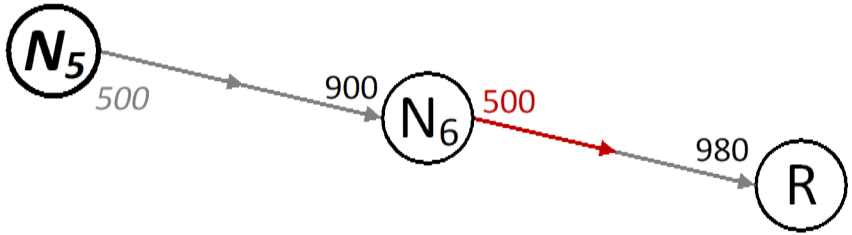
Insert N_6:

R	980



Insert N_5:

N_6	500
R	490



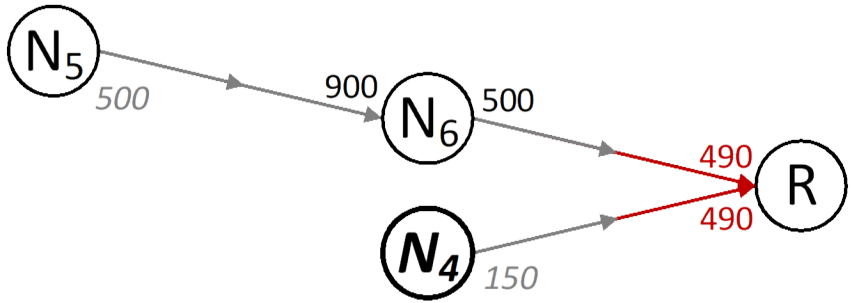
Algorithm Example

	N_3	N_5	N_4	N_5	N_6	R
Uplink (Mbps)	750	500	150	500	500	980
Downlink (Mbps)	100	130	1000	200	900	980

- **Step 1 (Inserting)**

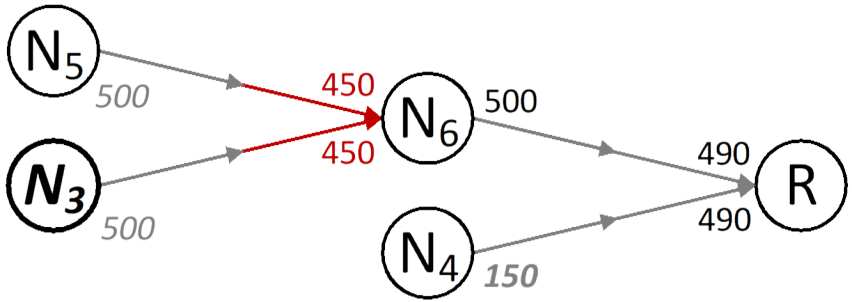
Insert N_4:

R	490
N_6	450
N_5	200



Insert N_3:

N_6	450
R	327
N_5	200
N_4	150



Algorithm Example

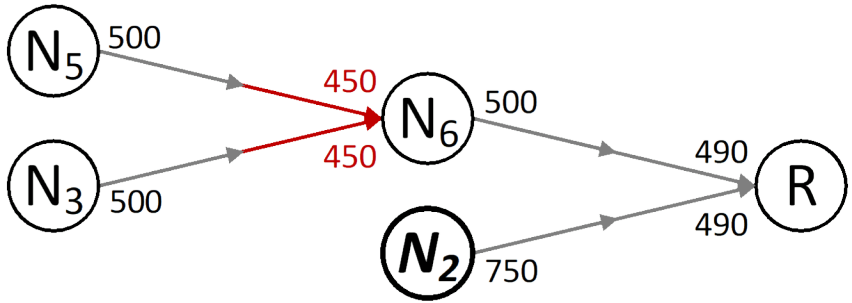
	N_2	N_3	N_4	N_5	N_6	R
Uplink (Mbps)	750	500	150	500	500	980
Downlink (Mbps)	100	130	1000	200	900	980

- **Step 2 (Replacing)**

Replace N_4 with N_2:

L^*: ***N_2***, N_3, N_5

L_{replaced} : N_4



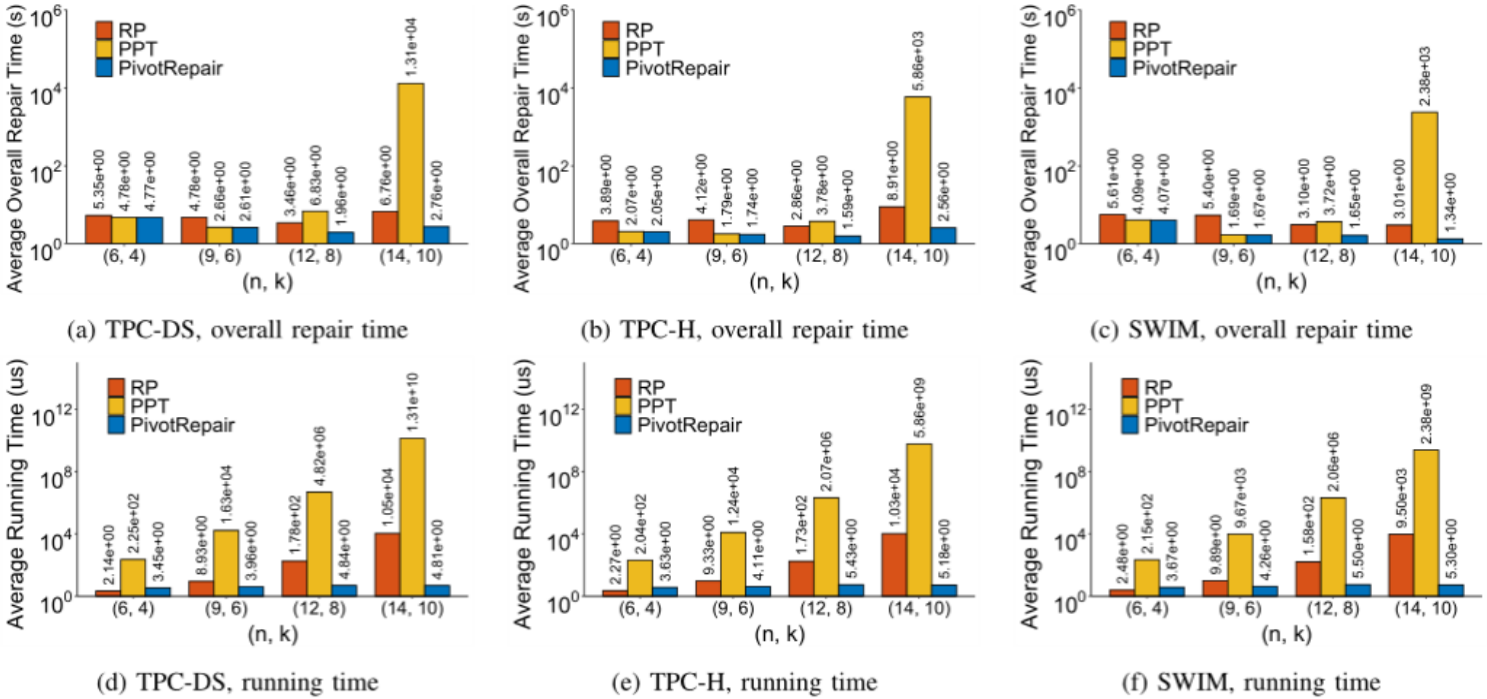
Optimal Pipelined Repair Tree

Full-Node Repair

- Limitations of the Straightforward Approach
  - Is impractical to generate optimal repair schemes for all stripes.
  - Fails to handle rapidly-changing network situations.

- Adaptive Scheduling
  - Focus: mitigates competition for bandwidth resources.
  - Idea: avoid starting a new task when its pipelined repair tree's links are shared by too many repair tasks in progress.
  - Recommendation value: selects the candidate repair task based on currently available bandwidths, the number of running tasks and delays of these tasks (compared to their estimated time).

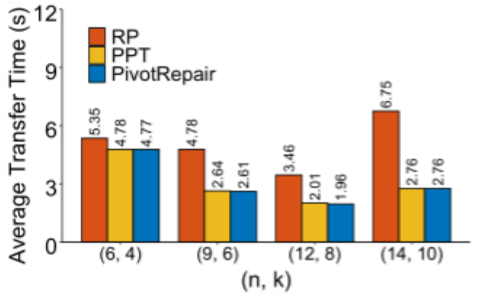
Evaluation



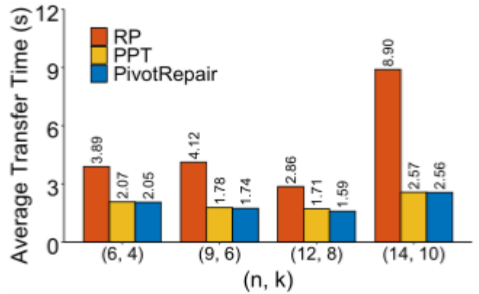
- Lower repair time than RP (by up to 71.27%).
- Much lower algorithm running time than PPT.

(see more details in the paper)

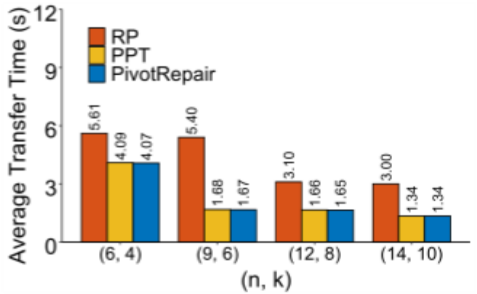
Evaluation



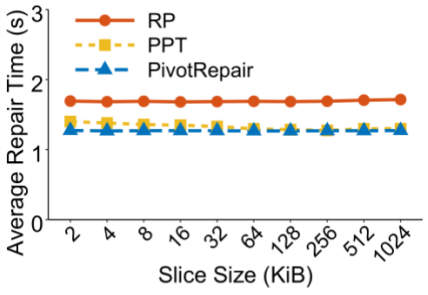
(g) TPC-DS, transfer time



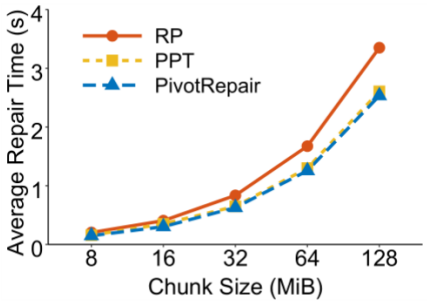
(h) TPC-H, transfer time



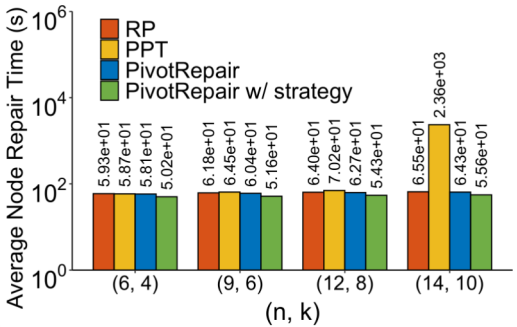
(i) SWIM, transfer time



(a) Impact of slice size



(b) Impact of chunk size



- Lower repair time than RP (by up to 71.27%).
- Much lower algorithm running time than PPT.

(see more details in the paper)

Conclusions

- We present **PivotRepair**, a fast pipelined repair technique for erasure-coded hot storage.
- We conduct measurement analysis and show that in hot storage clusters, congestion is frequent and rapidly changing, while some nodes still have abundant bandwidth.
- We present an optimal algorithm to construct quickly the pipelined repair tree by exploiting uncongested nodes called **pivots**, and propose an adaptive scheduling strategy to improve full-node repair performance.
- We prototype and evaluate PivotRepair on Amazon EC2. The evaluation demonstrates its efficiency in single-chunk and full-node repairs.

Source code: https://github.com/yuchonghu/PivotRepair

Thank You!

Contact:
Yuchong Hu yuchonghu@hust.edu.cn