Cocytus

Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication

AUTHORS: HENG ZHANG, MINGKAI DONG, HAIBO CHEN

Presenter: Kush Maheshwari

Key Ideas/Techniques

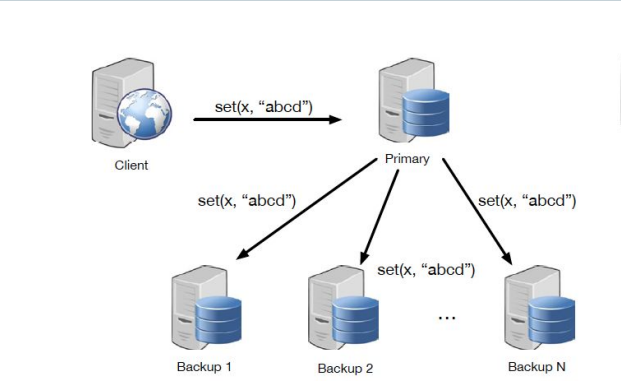
1. Current Problem and Solution
2. Exploiting Erasure Coding for in-memory KV store
3. Hybrid Replication (PBR + EC)
4. Distributed Online Recovery
5. Implementation
6. Evaluation of Cocytus against PBR
7. Discussion/Thoughts

The Demand

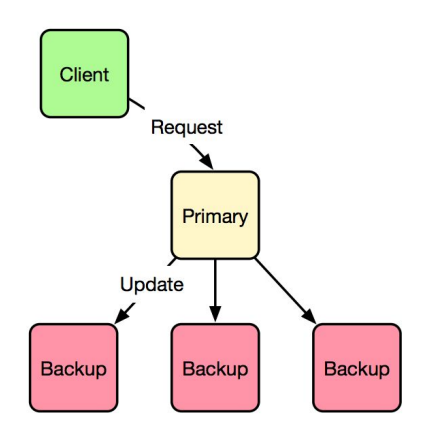
- Large scale web applications are demanding in-memory databases
  - Key building blocks for systems
- Availability and Efficiency for millions of requests per second
- Recovering 120 GB from disk to in-memory database took 2.5-3 hrs ~ Facebook study

Primary-backup Replication (PBR)

- Most common way to achieve availability
- Problems
  - Primary-backup requires M+1 replications to tolerate M failures
  - This wastes M copies of CPU/Memory that could be otherwise useful for other tasks

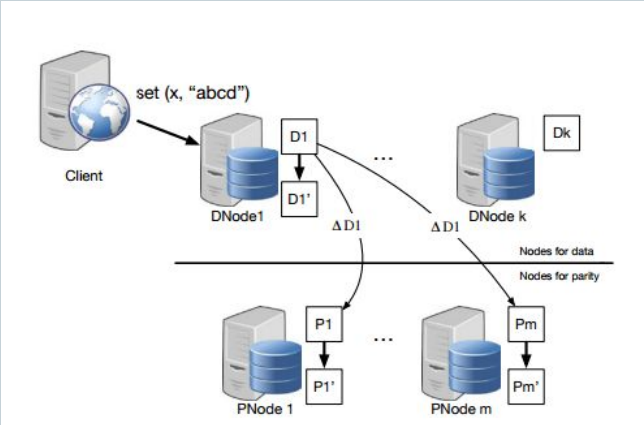


(a) Key/value store with primary-backup replication

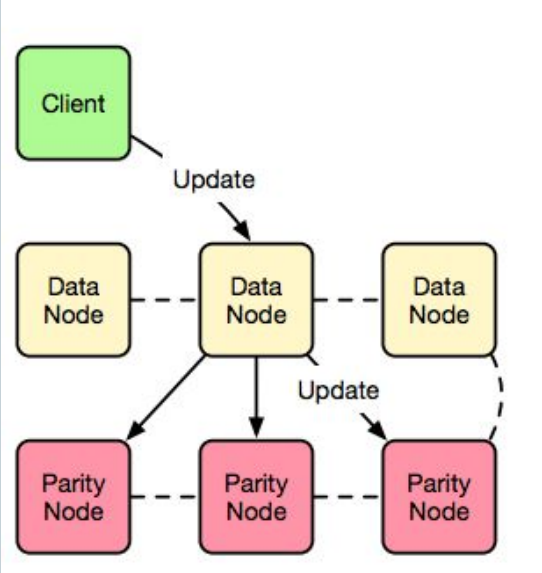


Erasure Coding

- A space-efficient way to prevent data loss
- An N node cluster can use K of N nodes for data and M nodes for parity (K+M=N)
- Uses Reed-Solomon coding scheme to compute parities



(b) Key/value store with erasure coding



Erasure Coding Math

- Vandermonde Matrix of predefined coefficients to calculate parities
- An update to a Dnode broadcasts its delta to all P nodes to recalculate parities

$$\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_M \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & a_1^1 & \cdots & a_1^{K-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & a_{M-1}^1 & \cdots & a_{M-1}^{K-1} \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_K \end{bmatrix}$$

- Any K nodes of the cluster can recover the original data by solving these equations
- Given RS(K,N) the system can handle M (N-K) failures

$$\begin{bmatrix} P_1' \\ \vdots \\ P_i' \\ \vdots \\ P_M' \end{bmatrix} = A * \begin{bmatrix} D_1 \\ \vdots \\ D_i' \\ \vdots \\ D_K \end{bmatrix} = A * \begin{bmatrix} D_1 \\ \vdots \\ D_i + \Delta D_i \\ \vdots \\ D_K \end{bmatrix} = \begin{bmatrix} P_1 \\ \vdots \\ P_i \\ \vdots \\ P_M \end{bmatrix} + \begin{bmatrix} 1 \\ \vdots \\ a_1^{i-1} \\ \vdots \\ a_{M-1}^{i-1} \end{bmatrix} * \Delta D_i$$

Opportunity

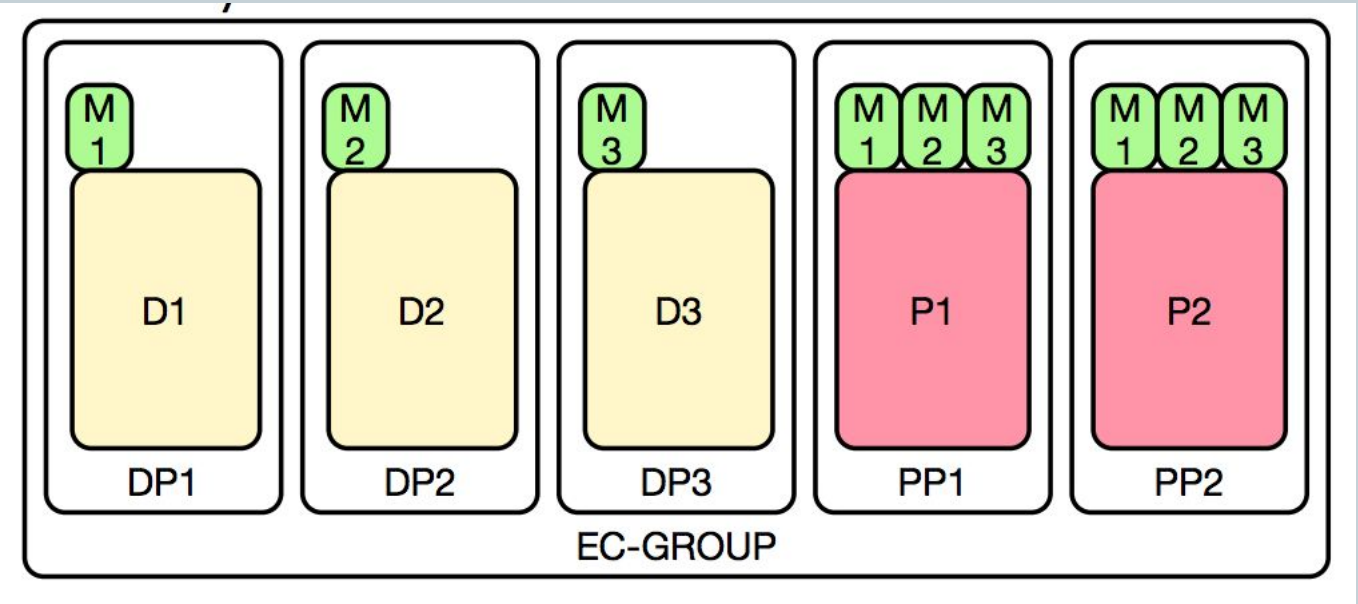
- CPU speed and CPU core counts are perfect for erasure coding
  - Encoding/Decoding rates are reaching 4.24-5.52 Gb/s on a single core
- Large network bandwidth reaching 40GB/s
- Trade CPU resources for memory efficiency and high availability

Reed-Solomon Coding Data 5-node Cluster

scheme	encoding speed	decoding speed
RS(4,5)	5.52GB/s	5.20GB/s
RS(3,5)	5.26GB/s	4.83GB/s
RS(2,5)	4.56GB/s	4.24GB/s

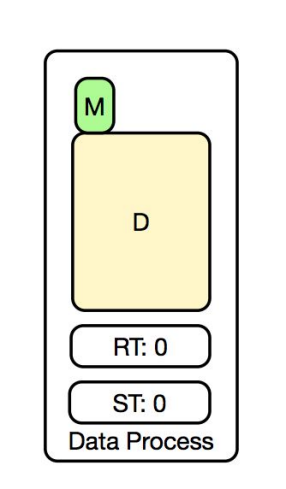
Cocytus Coding Group

- Coding groups are building blocks of Cocytus
  - Consists of K data processes and M parity processes
  - Connected by TCP

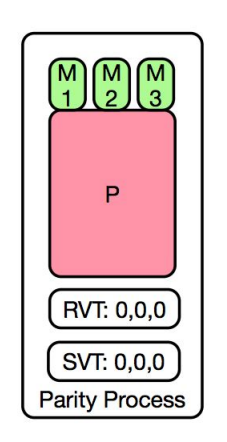


Data/Parity Process

- Metadata
  - Information of underlying data structure (index/allocation)
- Data Area
  - Where values are stored
- Logical Timestamps
  - Timestamp for latest Received Set request (RT)
  - Timestamp for latest Stable SET request (ST)

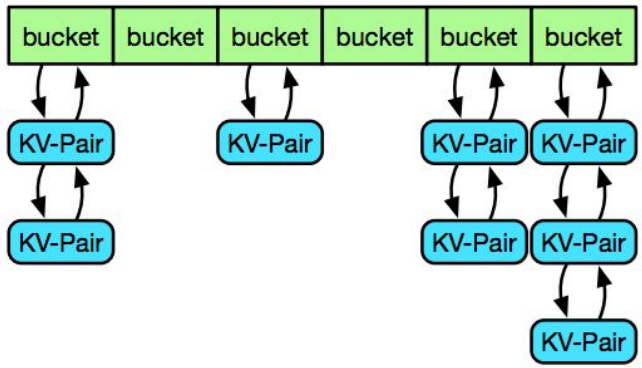


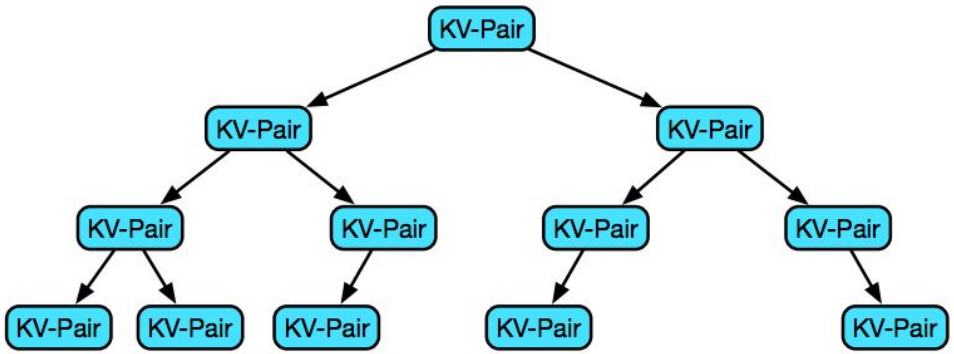
- Metadata replicas
  - PBR replica of metadata from every data process
- Parity area
  - Where parity data is stored
- Logical Timestamps
  - A Timestamp vector for latest Received SET requests(vector is # of data processes)
  - A Timestamp vector for latest Stable SET requests



Challenge: Excessive Update on Metadata

- Metadata is usually represented through either a hash table or a BST
  - These data structures need many operations when updated
- Operations on metadata involve allocating memory, freeing memory, inserting items into buckets, resizing hash tables





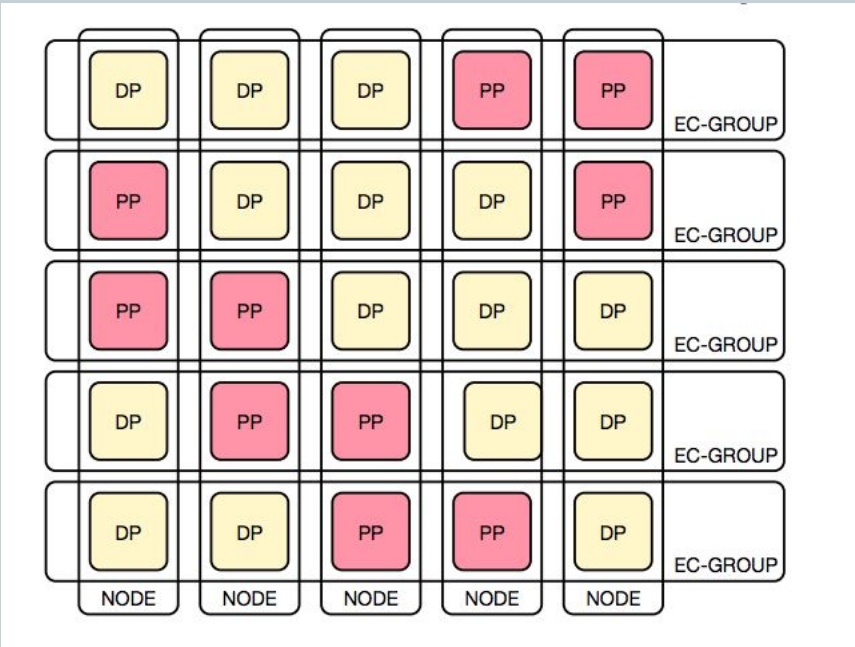
Solution: Separate Metadata from Data

- Erasure coding is not a good tool for metadata
  - Very complicated design
  - Excessive amount of encoding/transferring that limits design and increases latency
- Use erasure coding for values to prevent data loss
  - Pre-allocate areas of virtual memory for data and parity
- Use primary-backup replication (PBR) to handle mapping information
  - Save metadata for all data processes
  - Placing PBR on parity process leads to memory imbalance

Interleaved Layout

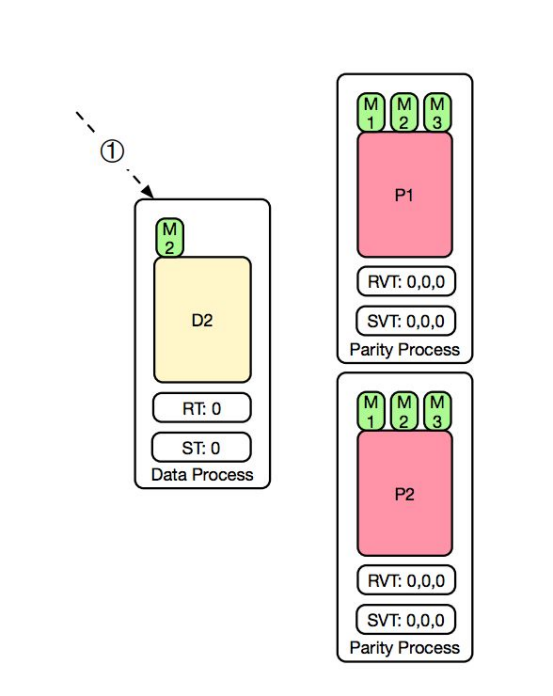
- If parity processes save more metadata than data processes there is a memory imbalance
- Don't want to leave parity processes idle or make them bottlenecks

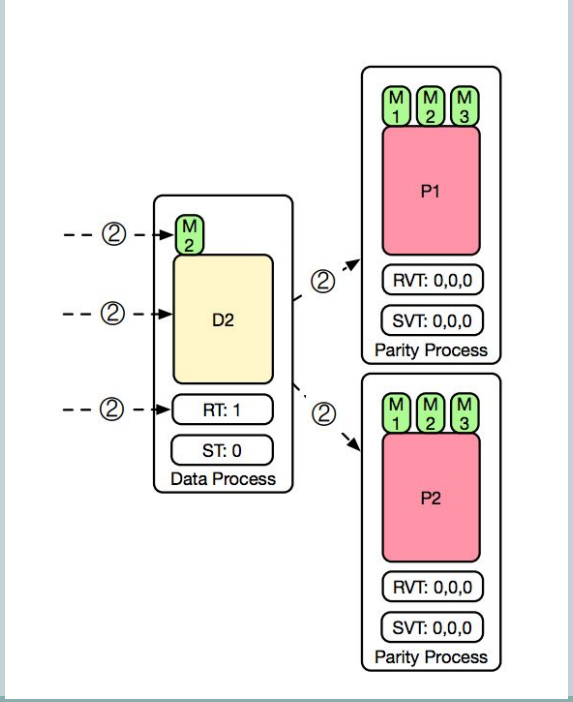
Solution: Interleave Data and Parity processes at different nodes so that ec-groups are spread out over the cluster



Handling a SET Request

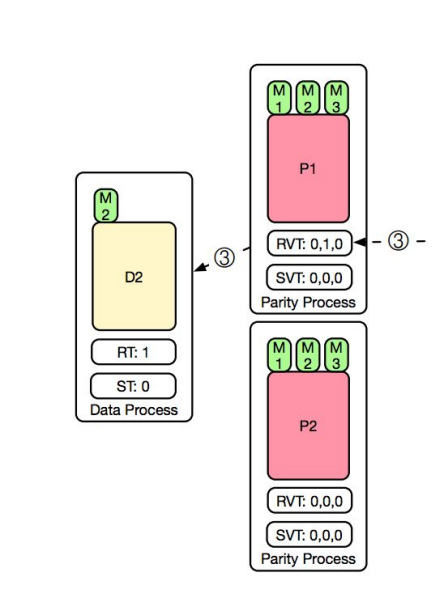
1. Send data to a Data Process
2. Handle request on data process
   1. Generate data diff
   2. Update timestamp

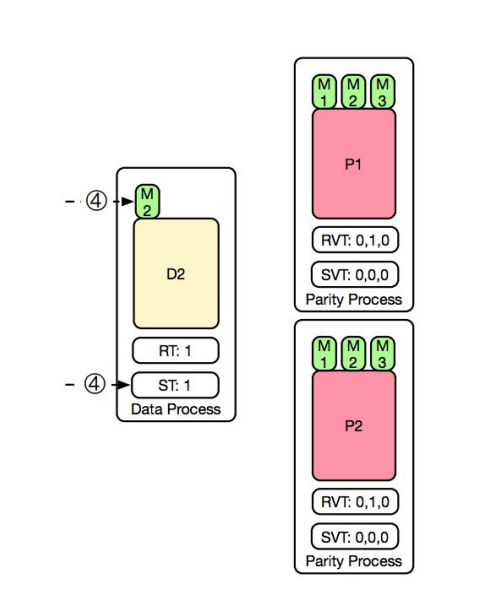




Handling a SET Request

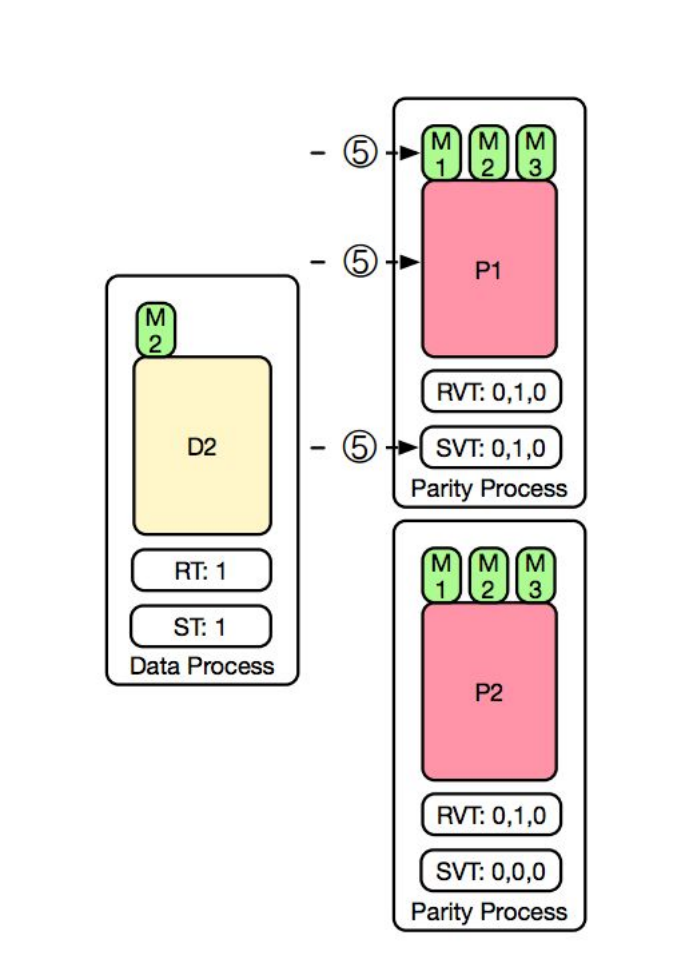
3. Handle request on Parity process
   1. Buffer request, update timestamp, send Acks
4. Complete request on data process
   3. Update in place, update timestamp, send commit request





Handling a SET request

5. Complete the request on parity process
   1. Update corresponding metadata
   2. Update parity area with diff
   3. Update SVT

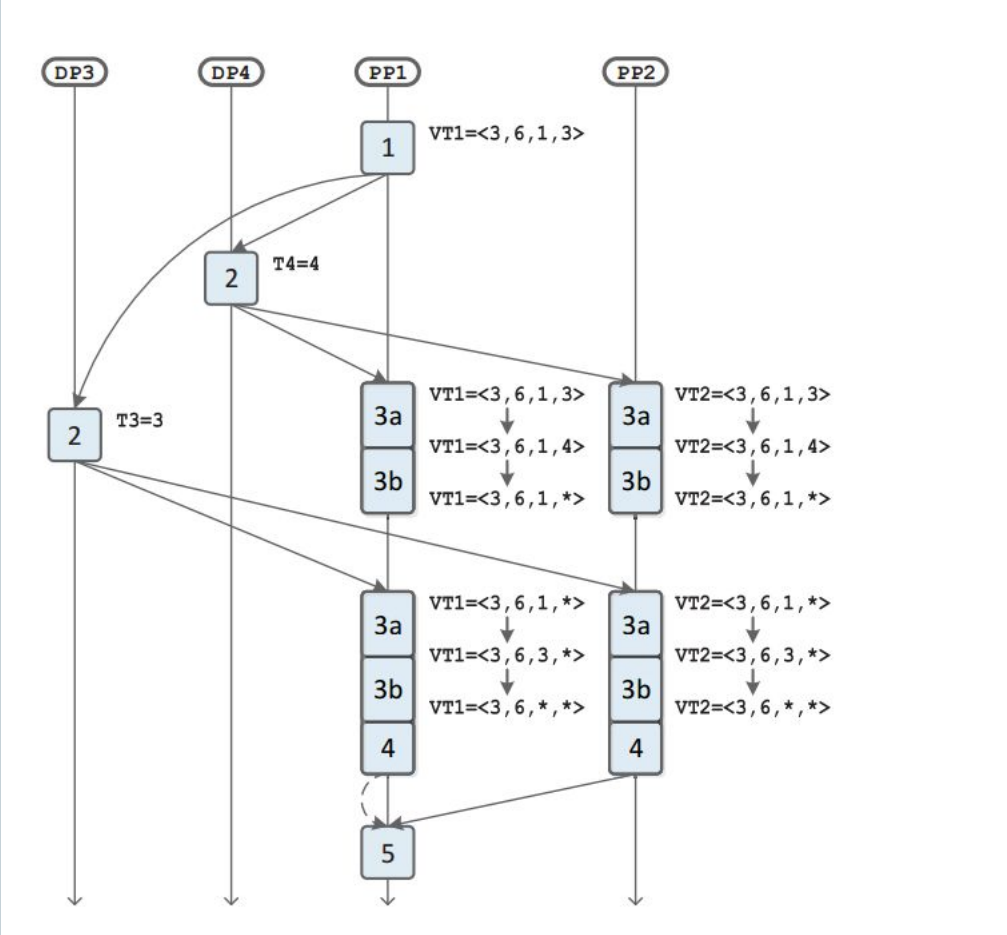


Data Recovery: Preparation Phase

1. Pick a corresponding Parity Process as a recovery process that also provides services
2. Collect latest Stable Ids corresponding to failed data node in all parity processes
3. Select minimum id and send to parity processes
4. Parity process apply all id requests <= min_id
5. Consistent Parity Processes with the failed data node

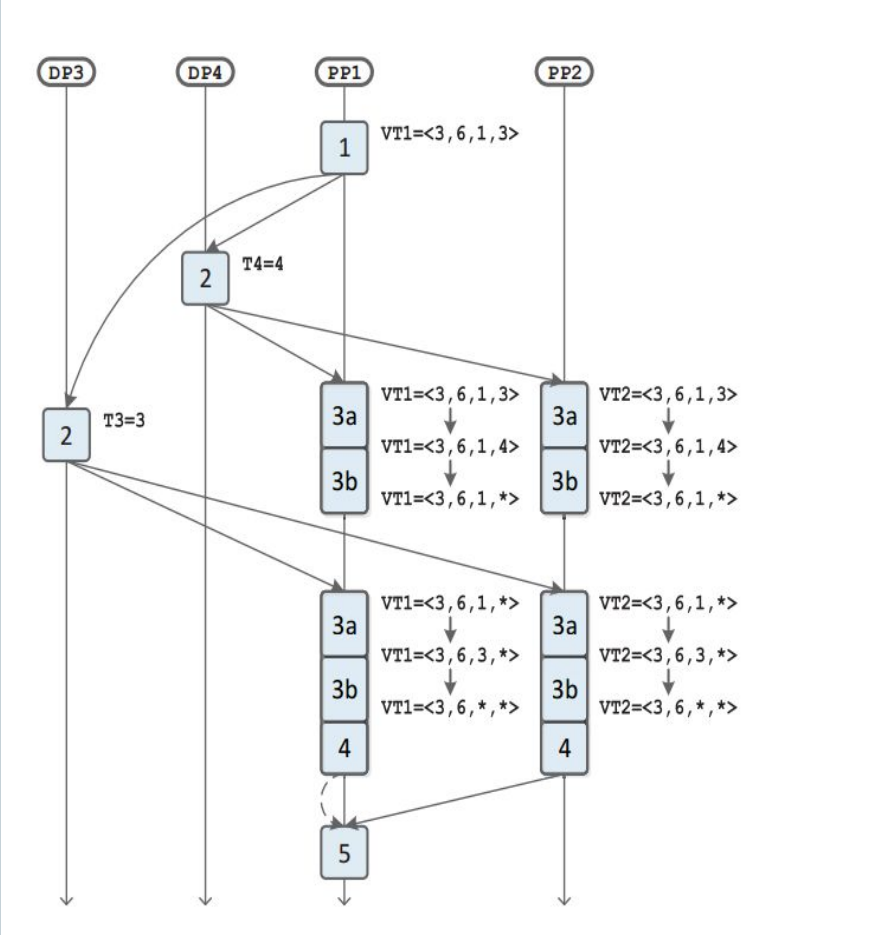
Online Recovery

1. A recovery leader is chosen from parity processes.
2. Sends recovery unit ID and alive processes to alive data processes



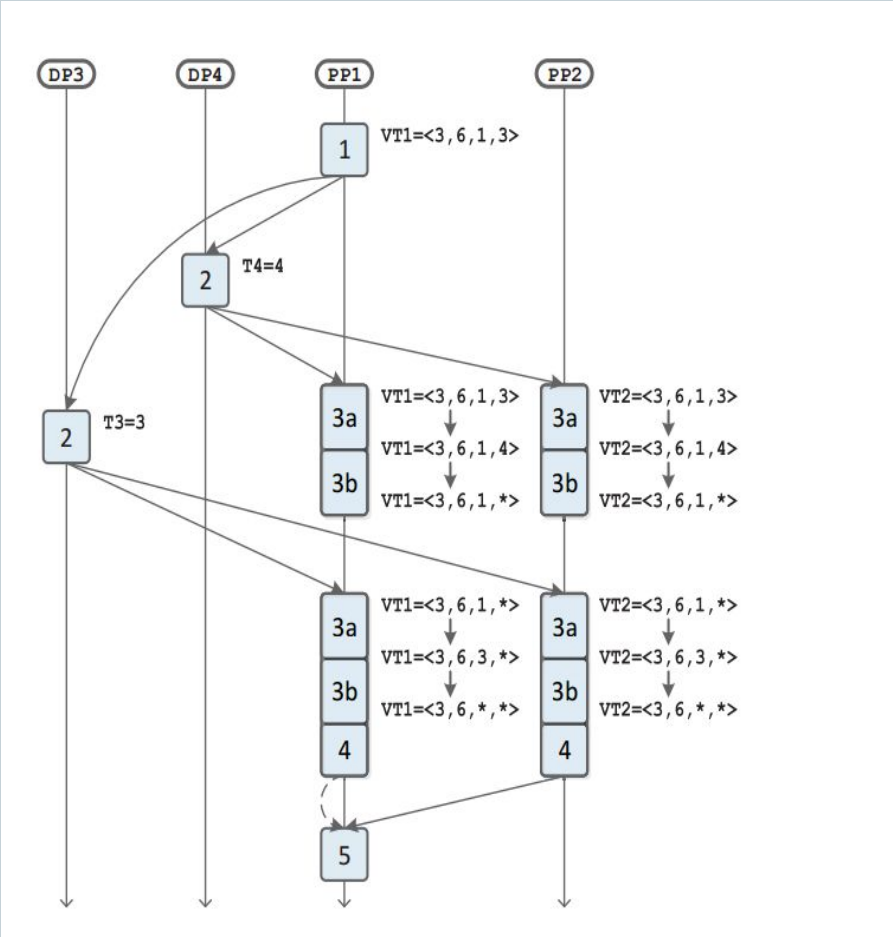
Online Recovery

1. Data Process sends data unit to recovery processes with id
2. Recovery process applies requests <= timestamp of data in its buffer
3. Subtracts parity unit from received data unit



Online Recovery

1. When all data units are are handled corresponding parity units are sent to the recovery initiator.
2. Decodes them using equations in Vandermonde Matrix



Request Handling on Recovery Process

- GET Request
  - Find pair through backup hashtable and check whether data block is recovered
  - If not recovered initiate data block recovery for that block and send back to client
- SET Request
  - Allocate new space for new value with allocation metadata
  - If allocated data blocks are not recovered call recovery function on them
  - After recovery handle operation like normal

Implementation

- Cocytus was implemented on top of Memcached
- Cocytus was also implemented with PBR in Memcached as a comparison
- Reed Solomon codes by Jerasure were used
- Deterministic/Pre-Alloc methods used

Evaluation Details

Cocytus

- 6-node cluster of machines
  - 5 servers and 1 client
  - Two 10 core 2.3 GHz Intel Xeon E5-2650, 64 GB of RAM
- 5 EC groups with 3 DPs and 2 PPs (25 in total)

PBR

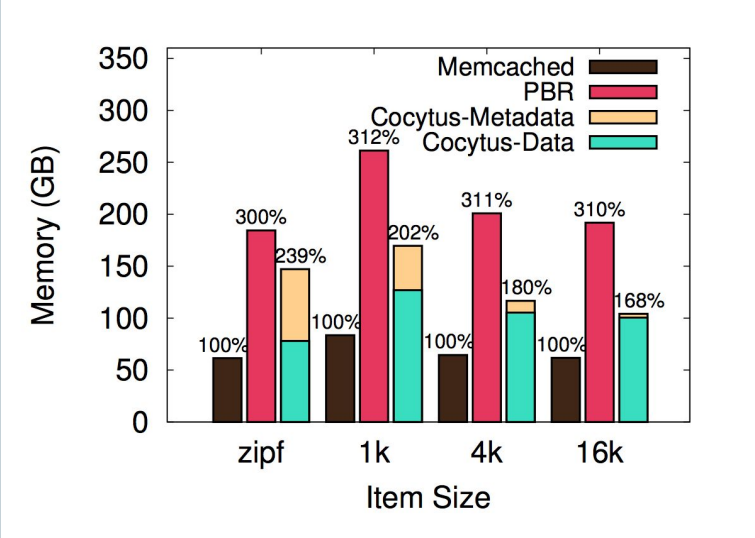
- 15 primary and 30 backup for PBR (45 in total)

Vanilla Memcached

- 15 instances among 5 nodes

Memory Consumption

- 16 KB values led to 46% memory saving
- Replicating metadata and keys was worse than they expected but cost is still worth in-memory recovery
- Zipf = distribution of 10B to 1KB values
  - 100 million items so more metadata
  - Still 20% memory reduction

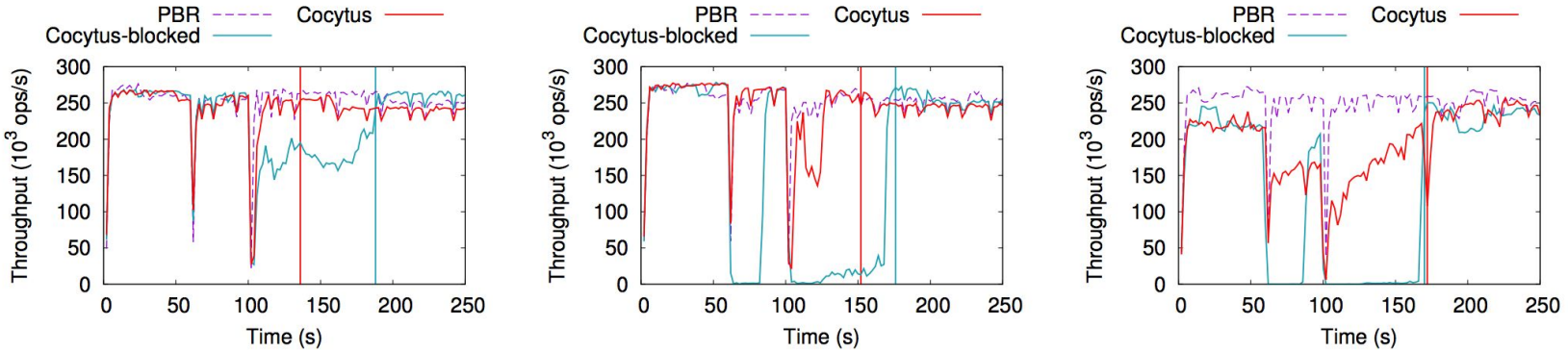


CPU Consumption

Read:Write	Memcached	PB Replication		Cocytus	
	15 processes	15 primary processes	30 backup processes	15 data processes	10 parity processes
50%:50%	231%CPUs	439%CPUs	189%CPUs	802%CPUs	255%CPUs
95%:5%	228%CPUs	234%CPUs	60%CPUs	256%CPUs	54%CPUs
100%:0%	222%CPUs	230%CPUs	21%CPUs	223%CPUs	15%CPUs

- CPU for read-write workloads is double PBR
  - To be expected given amount of data written
  - Main point is that Memory is conserved

Recovery Efficiency



(a) Throughput on recovery (r:w = 100:0) (b) Throughput on recovery (r:w = 95:5) (c) Throughput on recovery (r:w = 50:50)

- Emulate 2 node failures at 60s and 100s
- After 1st node failure Cocytus can repair data at 550 MB/s
  - Data recovery can be done in parallel with read requests

Evaluation

- Reduces memory consumption by 33% to 46% for value sizes from 1KB to 16KB when tolerating 2 node failures
- Low overhead: Similar throughput as PBR
- Under 2 node crashes, Cocytus still was able to handle client requests as well as PBR and gracefully recover data

Discussion/Thoughts

- Me
  - How does Cocytus know which node to look at for GET/SET
  - How does it deal with 2 failures in a row?
- Class
  - Poor evaluation-does not consider scale
  - Deploying too many data processes in 1 group increases burden on parity process
  - Byzantine/Power outage not taken into account
  - CPU for Memory (especially with a changing KV store)
  - How does it deal with client requests during recovery?