Cocytus

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

Heng Zhang, Mingkai Dong, Haibo Chen

Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University, China

http://ipads.se.sjtu.edu.cn/pub/projects/cocytus
In-memory KV-Stores: Key Building Blocks for Systems

• A key pillar for many systems
  • Data cache (e.g., Memcached in Facebook)
  • In-memory database

• Availability is important for in-memory KV-Stores
  • Services disruption
  • Recovery is time-consuming
Primary-backup Replication (PBR)

• A common way to achieve availability
  • E.g., Repcached, Redis

• Problems
  • Need $M$ times extra memory to tolerate $M$ failures
  • Redundant data is rarely accessed in strongly consistent systems
Erasure Coding (EC)

• A space-efficient way to prevent data loss

• Widely used in disk storage
  • RAID (Redundant Array of Independent Disks)
  • WAS (Windows Azure Storage)

• Data repair needs to collect data and decode them
  • A lot of computing work and data transfer
Opportunity

• Large network bandwidth
  • Reaches 10Gb/s and 40Gb/s

• Fast speed of CPUs
  • Encoding/Decoding rates can also reach 40Gb/s on single core
Goal

Erasure Coding + In-memory KV-Stores

⇒

Available and Memory Efficient
In-memory KV-stores
Intuited System Design

- K nodes for storing data
- M nodes for storing parity
- Each key-value pair is totally stored on one data node
  - friendly for GET requests
Challenges

• Excessive metadata update

• Race condition in online recovery
Excessive Update on Metadata

• Metadata is usually achieved by scattered and linked data structure
  • E.g., hash table and binary search tree (BST), two popular data structures for in-memory index
Excessive Update on Metadata

• Metadata is usually achieved by scattered and linked data structure

• Operations on metadata involve many scattered modifications
  • About 4 scattered modifications on allocating memory
  • About 7 scattered modifications on freeing memory
  • About 4 scattered modifications on inserting new item into bucket hash table
  • $O(N)$ scattered modifications on resizing of hash table
Excessive Update on Metadata

• Metadata is usually achieved by scattered and linked data structure
• Operations on metadata involve many scattered modifications
• Erasure coding is not a good choice for metadata
  • Complicated implementation
  • A SET request involve encoding/transfer for 7-14 scattered changes
  • Limit new metadata design
Solution: Separate data and metadata

• Use **erasure coding** to prevent data (values) loss  
  • Pre-allocate virtual memory areas for data and parity  
  • Modifications on these areas agree with erasure coding approach

• Use **primary-backup replication** to prevent metadata loss  
  • Index information and allocation information are placed on outside of the area
Race Condition in Online Recovery

- Handle GET/SET requests during recovery
- Handling SET request involves update on multiple nodes
- Data repair needs to collect data and parity among nodes
Race Condition in Online Recovery

- The interleaving of SET requests and data repair has race condition

* The $f_1()$ and $f_2()$ is encoding functions
* The $g_1()$ is a decoding function
* The collected parity $P_1$ loses the new update

```
Client

(Failed)
Data
Node
D_1=2

① SET D_2=5

Data
Node
D_2=1

② D_2=5

Data
Node
D_3=3

② D_3=3

Parity
Node
P_1=f_1(2,1,3)

③ P_1=f_1(2,1,3)

Parity
Node
P_2=f_2(2,1,3)

③
data
Node
D_1=2

Recover
Data

D_1 = g_1(f_1(2,1,3), 5, 3) != 2
```
Online Recovery Protocol

• Use logical timestamp to indicate the version of data
  • Attach timestamps on SET requests
  • In-order completion

• Three steps for data collection
  1. Start procedure
  2. Decide data versions
  3. Synchronize parity version
Cocytus Overview
EC-Group

- EC-Group is the basic component in Cocytus
  - A EC-Group consists $K$ data processes and $M$ parity processes
  - Connected by a FIFO channel like a TCP connection
Data Process

• Metadata
  • Index information
  • Allocation information

• Data area
  • A memory area for values

• Logical timestamps
  • A Timestamp for the latest Received SET request (RT)
  • A Timestamp for the latest Stable (completed) SET request (ST)
Parity Process

• Metadata replicas of all data processes in the EC-Group

• Parity area
  • A memory area for parity

• Logical timestamps
  • A Timestamp Vector for the latest Received SET requests (RVT[1..K])
  • A Timestamp Vector for the latest Stable (completed) SET requests (SVT[1..K])
Workloads Imbalance

• Data processes and parity processes have different work
• Data processes and parity processes reserve memory in different size

• Solution: interleaved layout
Handling SET Requests
Handling a SET Request

1. Dispatch to a data process
Handling a SET Request

1. Dispatch to a data process

2. Handle the request on the data process
   1. Generate data diff
   2. Update the timestamp
   3. Forward request
Handling a SET Request

1. Dispatch to a data processes
2. Handle the request on the data process
3. Handle the request on parity processes
   1. Buffer the request
   2. Update the timestamps
   3. Send ACKs
Handling a SET Request

2. Handle the request on the data process
3. Handle the request on parity processes
4. Complete the request on the data process
   1. Update in place
   2. Update the timestamp
   3. Send commit requests
Handling a SET Request

3. Handle the request on parity processes
4. Complete the request on the data process
5. Complete the request on parity processes
   1. Update corresponding metadata
   2. Update parity area with diff
   3. Update SVT
Recovery
Online Recovery

• When a data process fails, Cocytus chooses a recovery process from parity processes
  • Start two-phases recovery
  • Provide continuously services

• Two-phases recovery
  • Preparation: synchronize parity processes
  • Online data repair: repair the data area while handling requests

• Choose a recovery leader on multiple failures
Preparation

- The recovery process synchronizes stable timestamp for the failed data process
  1. collect corresponding $RVT[i]$s from all parity processes, where $i$ is the failed data node

  After preparation phase, all parity processes are consistent in the failed data process

- Parity processes complete the buffered requests that
  - contain equal or smaller timestamps than the synchronized stable timestamp
  - come from the failed data processes
Preparation

• The recovery process synchronizes stable timestamp for the failed data process
  1. collect corresponding RVT[i]s from all parity processes, where i is the failed data node
  2. choose the minimal one to be the synchronized stable timestamp
  3. broadcast the synchronized stable timestamp to other parity processes

• Parity processes complete the buffered requests that
  • contain equal or smaller timestamps than the synchronized stable timestamp
  • come from the failed data processes
Online Data Repair

• Data area is repaired in a granularity of 4KB page

• Page repair happens
  • When requests need touch a lost page
  • In the background

• Under online recovery protocol
Recovery Protocol

Recovery leader
1. Choose the parity participant
2. Notify alive data processes
Recovery Protocol

Data processes
1. Decide stable timestamp
2. Send data page

Parity processes
1. Synchronize the stable timestamps
2. Do partial decoding
Recovery Protocol

Parity processes
1. send partially decoded parity
Recovery Protocol

Recovery leader

1. Complete the decoding
2. Send recovered data pages to other recovery processes
Implementation

• Cocytus is implemented on Memcached 1.4.21
  • Implement a similar primary-backup replication version for comparison

• Coding Scheme
  • Reed-Solomon code provided by Jerasure
Evaluation

• 5-node cluster for server
  • 5 EC-Groups for Cocytus, each contains 3 DPs and 2 PPs
  • 15 primary processes and 30 backup processes for primary-backup replication version
  • 15 original processes for Memcached

• 1 node for client, 20 cores
  • Run YCSB benchmark with 80 threads

• 10Gbps network
Memory Consumption

*ZIPF: Zipfian distribution over the range from 10B to 1KB
Recovery

(R:W=95%:5% & 1KB-size value & 12GB data/node)
## CPU Overhead

<table>
<thead>
<tr>
<th>Read:Write</th>
<th>Memcached</th>
<th>PB Replication</th>
<th>Cocytus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15 processes</td>
<td>15 primary processes</td>
<td>15 data processes</td>
</tr>
<tr>
<td></td>
<td>30 backup processes</td>
<td>10 parity processes</td>
<td></td>
</tr>
<tr>
<td>50%:50%</td>
<td>231%CPUs</td>
<td>439%CPUs</td>
<td>802%CPUs</td>
</tr>
<tr>
<td>95%:5%</td>
<td>228%CPUs</td>
<td>234%CPUs</td>
<td>256%CPUs</td>
</tr>
<tr>
<td>100%:0%</td>
<td>222%CPUs</td>
<td>230%CPUs</td>
<td>223%CPUs</td>
</tr>
</tbody>
</table>

- **Memcached**: 15 processes
- **PB Replication**: 30 backup processes
- **Cocytus**: 10 parity processes
Related Work

• Separation of work
  • Gnothi\textsuperscript{ATC'12}, UpRight\textsuperscript{SIGOPS'09} ...

• Erasure coding
  • WAS\textsuperscript{ATC'12}, XORing Elephants\textsuperscript{VLDB'13} ...

• Replication
  • Mojim\textsuperscript{ASPLOS'15}, RAMCloud\textsuperscript{SOSP'11} ...

• Key-value stores
  • Pilaf\textsuperscript{ATC'13}, FaRM\textsuperscript{NSDI'14}, HERD\textsuperscript{SIGCOMM'14}, and C-Hint\textsuperscript{SoCC'14} ...
Conclusion

• Replication approach is quit memory-consuming for in-memory KV-Stores

• Cocytus combines erase coding and replication to achieve efficient and available in-memory KV-Store

• Cocytus could achieve better memory efficiency with low overhead compared with primary-backup replication on read-mostly workloads

Thanks

http://ipads.se.sjtu.edu.cn