Giza: Erasure Coding Objects across Global Data Centers

Yu Lin Chen*, Shuai Mu , Jinyang Li , Cheng Huang *, Jin li *, Aaron Ogus *, and Douglas Phillips*
* New York University, *Microsoft Corporation
USENIX ATC, Santa Clara, CA, July 2017
Azure data centers span the globe
Data center fault tolerance: Geo Replication
Data center fault tolerance: Erasure coding

Project Giza
The cost of erasure coding

Project Giza
When is cross DC coding a win?

1. Inexpensive cross DC communication.
2. Coding friendly workload
   a) Stores a lot of data in large objects (less coding overhead)
   b) Infrequent reconstruction due to read or failure.
Cross DC communication is becoming less expensive.

Largest bandwidth capacity today with 160Tpbs.
In search of a coding friendly workload

3 months trace: January – March 2016
OneDrive is occupied mostly by large objects
OneDrive is occupied mostly by large objects

![CDF vs Object Size](chart.png)

Object > 1GB occupies 53% of total storage capacity.
OneDrive is occupied mostly by large objects.
OneDrive reads are infrequent after a month
OneDrive reads are infrequent after a month

47% of the total bytes read occur within the first day of the object creation.
OneDrive reads are infrequent after a month

less than 2% of the total bytes read occur after a month
Outline

1. The case for erasure coding across data centers.

2. Giza design and challenges

3. Experimental Results
Giza: a strongly consistent versioned object store

• Erasure code each object individually.

• Optimize latency for Put and Get.

• Leverage existing single-DC cloud storage API.
Giza’s architecture
Giza’s workflow

Client

Put(Blob1, Data)

Giza node

Table

Blob

Giza node

Giza node

Giza node

DC1

Giza node

Giza node

Giza node

DC2

Giza node

Giza node

Giza node

DC3
Giza’s workflow

Client

Put(Blob1, Data)

Put(GUID1, Fragment A)

Put(GUID2, Fragment B)

Put(GUID3, Fragment C)

DC₁

DC₂

DC₃
Giza’s workflow

Client

Put(Blob1, Data)

{DC1: GUID1, DC2: GUID2, DC3: GUID3}

DC1

Table

Blob

DC2

Giza node

Table

Blob

Giza node

DC3

Giza node

Table

Blob
Giza metadata versioning

Let $\text{md}_{\text{this}} = \{\text{DC}_1: \text{GUID}_4, \text{DC}_2: \text{GUID}_5, \text{DC}_3: \text{GUID}_6\}$

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata (version #: actual metadata)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: $\text{md}_1$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata (version #: actual metadata)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: $\text{md}<em>1$, v2: $\text{md}</em>{\text{this}}$</td>
</tr>
</tbody>
</table>
Inconsistency during concurrent put operations

Client

Put(Blob1, Data1)

Giza node

Giza node

Giza node

Table

Blob

DC1

Client

Put(Blob1, Data2)

Giza node

Giza node

Giza node

Table

Blob

DC2

Table

Blob

DC3
Inconsistency during concurrent put operations

Order of arrival

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
<th>DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: $md_1$</td>
<td>DC$_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
<th>DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: $md_2$</td>
<td>DC$_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
<th>DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: $md_1$</td>
<td>DC$_3$</td>
</tr>
</tbody>
</table>

WARNING: INCONSISTENT STATE
Inconsistency during concurrent put operations

Order of arrival

Key | Metadata
---|---
Blob1 | v1: \( md_1 \)

DC_1

Key | Metadata
---|---
Blob1 | v1: \( md_2 \)

DC_2

Key | Metadata
---|---
Blob1 | v1: \( md_1 \)

DC_3

WARNING: INCONSISTENT STATE
Choosing version value via consensus

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₁</td>
</tr>
<tr>
<td></td>
<td>v2: md₂</td>
</tr>
<tr>
<td>DC₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₁</td>
</tr>
<tr>
<td></td>
<td>v2: md₂</td>
</tr>
<tr>
<td>DC₂</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₂</td>
</tr>
<tr>
<td></td>
<td>v2: md₁</td>
</tr>
<tr>
<td>DC₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₂</td>
</tr>
<tr>
<td></td>
<td>v2: md₁</td>
</tr>
<tr>
<td>DC₂</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₁</td>
</tr>
<tr>
<td></td>
<td>v2: md₂</td>
</tr>
<tr>
<td>DC₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₁</td>
</tr>
<tr>
<td></td>
<td>v2: md₂</td>
</tr>
<tr>
<td>DC₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₂</td>
</tr>
<tr>
<td></td>
<td>v2: md₁</td>
</tr>
<tr>
<td>DC₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₁</td>
</tr>
<tr>
<td></td>
<td>v2: md₂</td>
</tr>
<tr>
<td>DC₂</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₂</td>
</tr>
<tr>
<td></td>
<td>v2: md₁</td>
</tr>
<tr>
<td>DC₃</td>
<td></td>
</tr>
</tbody>
</table>
Paxos and Fast Paxos properties

Paxos

• Can commit with 2 round trips.

• Requires majority of replicas to commit.

\[ \frac{N}{2} + 1 \]
## Paxos and Fast Paxos properties

<table>
<thead>
<tr>
<th>Paxos</th>
<th>Fast Paxos</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Can commit with 2 round trips.</td>
<td></td>
</tr>
<tr>
<td>• Requires majority of replicas to commit. ( \frac{N}{2} + 1 )</td>
<td></td>
</tr>
<tr>
<td>• Can commit with 1 round trip.</td>
<td></td>
</tr>
<tr>
<td>• Requires more than the majority of replicas to commit. ( \frac{2N}{3} + 1 )</td>
<td></td>
</tr>
</tbody>
</table>
Implementing Fast Paxos with Azure table API

Let $md_{\text{this}} = \{\text{DC}_1: \text{GUID}_1, \text{DC}_2: \text{GUID}_2, \text{DC}_3: \text{GUID}_3\}, \text{Paxos states}\)
Implementing Fast Paxos with Azure table API

Let $md_{this} = \{DC_1: GUID_1, DC_2: GUID_2, DC_3: GUID_3\}$, Paxos states)
Implementing Fast Paxos with Azure table API

Let $md_{\text{this}} = \{\text{DC}_1: \text{GUID}_1, \text{DC}_2: \text{GUID}_2, \text{DC}_3: \text{GUID}_3\}, \text{Paxos states}\)

Read metadata row and decide to accept or reject proposal ($v2 = md_{\text{this}}$)

E tag = 1
Implementing Fast Paxos with Azure table API

Let \( \text{md}_{\text{this}} = \{\text{DC}_1:\text{GUID}_1, \text{DC}_2:\text{GUID}_2, \text{DC}_3:\text{GUID}_3\}, \text{Paxos states} \)

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata (version #: actual metadata)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md_1, v2: md_2</td>
</tr>
</tbody>
</table>

\[ \text{E tag} = 1 \]

Fail to update table since etag has changed

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata (version #: actual metadata)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md_1, v2: md_2</td>
</tr>
</tbody>
</table>

\[ \text{E tag} = 2 \]

Remote DC

Write metadata row back to the table if proposal accepted
Implementing Fast Paxos with Azure table API

Let \( md_{\text{this}} = \{\text{DC}_1: \text{GUID}_1, \text{DC}_2: \text{GUID}_2, \text{DC}_3: \text{GUID}_3\}, \) Paxos states)

E tag = 1

Successfully update table since etag hasn’t changed

Write metadata row back to the table if proposal accepted
Parallelizing metadata and data path

DC1

Giza node
Giza node
Giza node

Table
Blob

DC2

Giza node
Giza node
Giza node

Table
Blob

DC3

Giza node
Giza node
Giza node

Table
Blob
Data path fails, metadata path succeeds

<table>
<thead>
<tr>
<th>Key</th>
<th>Metadata (version #: actual metadata)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>v1: md₁, v2: md₂</td>
</tr>
</tbody>
</table>

Ongoing, data path failed

• During read, Giza will revert back to the latest version where the data is available.
Giza Put

- Execute metadata and data path in parallel.
- Upon success, return acknowledgement to the client.
- In the background, update highest committed version number in the metadata row.

<table>
<thead>
<tr>
<th>Key</th>
<th>Committed</th>
<th>Metadata (version #: actual metadata)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob1</td>
<td>1</td>
<td>v1: md_1, v2: md_2</td>
</tr>
</tbody>
</table>

Ongoing, not committed
Giza Get

• Optimistically read the latest committed version locally.

• Execute both data path and metadata path in parallel:
  • Data path retrieves the data fragments indicated by the local latest committed version.
  • Metadata path will retrieve the metadata row from all data centers and validate the latest committed version.
Outline

1. The case for erasure coding across data centers.

2. Giza design and challenges

3. Experimental Results
Giza deployment configurations
Giza deployment configurations
Giza deployment configurations
Giza deployment configurations
Metadata Path: Fast vs Classic Paxos

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th>Query Version Latency</th>
<th>Table Latency</th>
<th>Transfer Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-100</td>
<td>Fast</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110-200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210-300</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>310-400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410-500</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Fast vs Classic comparison
- Latency measurements: 23ms, 32ms, 64ms
Metadata Path: Fast vs Classic Paxos

- Query Version Latency
- Table Latency
- Transfer Latency

Latency (ms)

- Fast: 102ms, 139ms
- Classic: 240ms, 139ms
Giza Put Latency (World-2-1)
Giza Get Latency (World-2-1)
Summary

• Recent trend in data centers supports the economic feasibility of erasure coding across data centers.

• Giza is a strongly consistent versioned object store, built on top of Azure storage.

• Giza optimizes for latency and is fast in the common case.