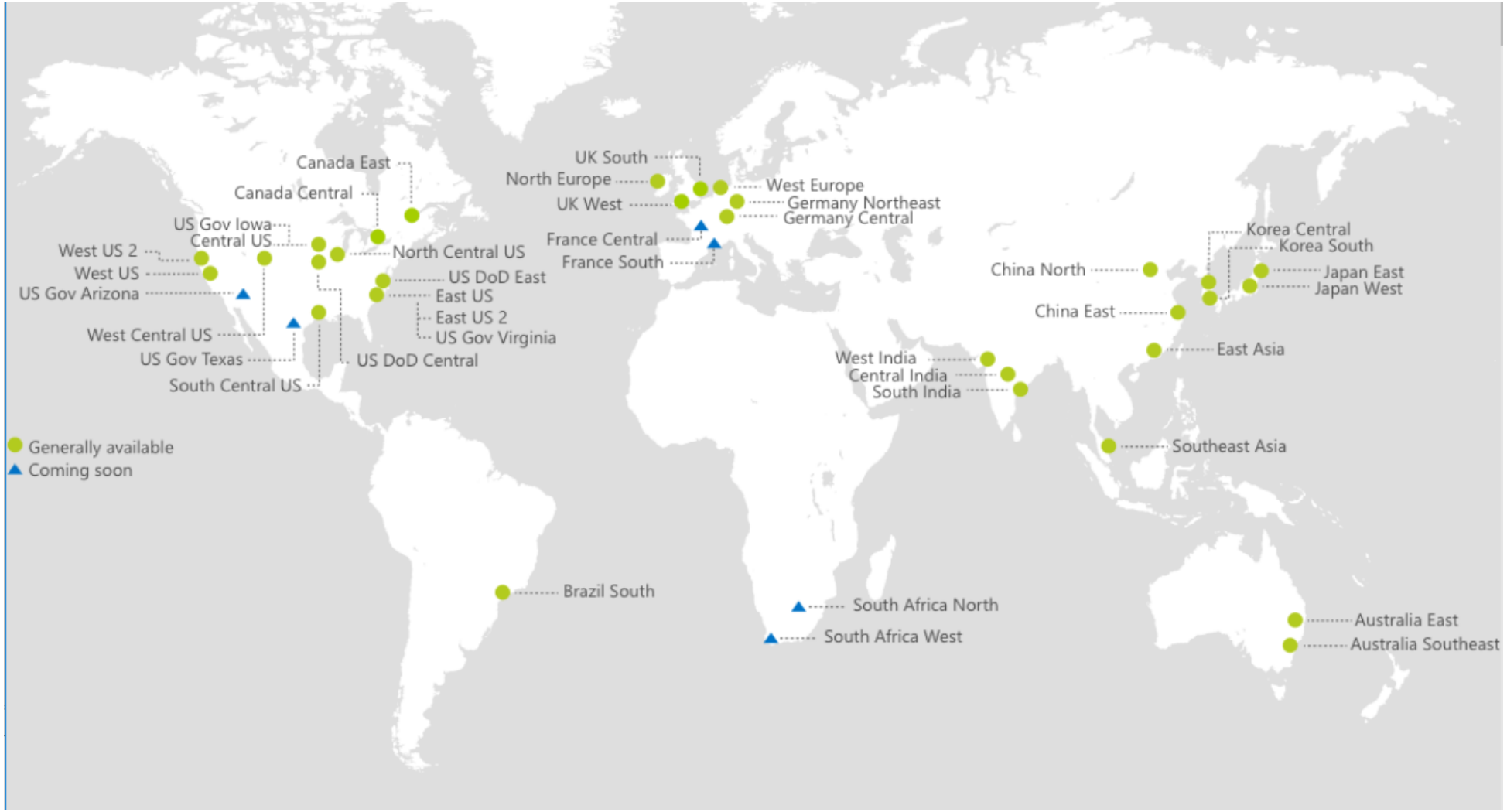
Giza: Erasure Coding Objects across Global Data Centers

Yu Lin Chen*, Shuai Mu , Jinyang Li , Cheng Huang *, Jin li *, Aaron Ogus *, and Douglas Phillips*

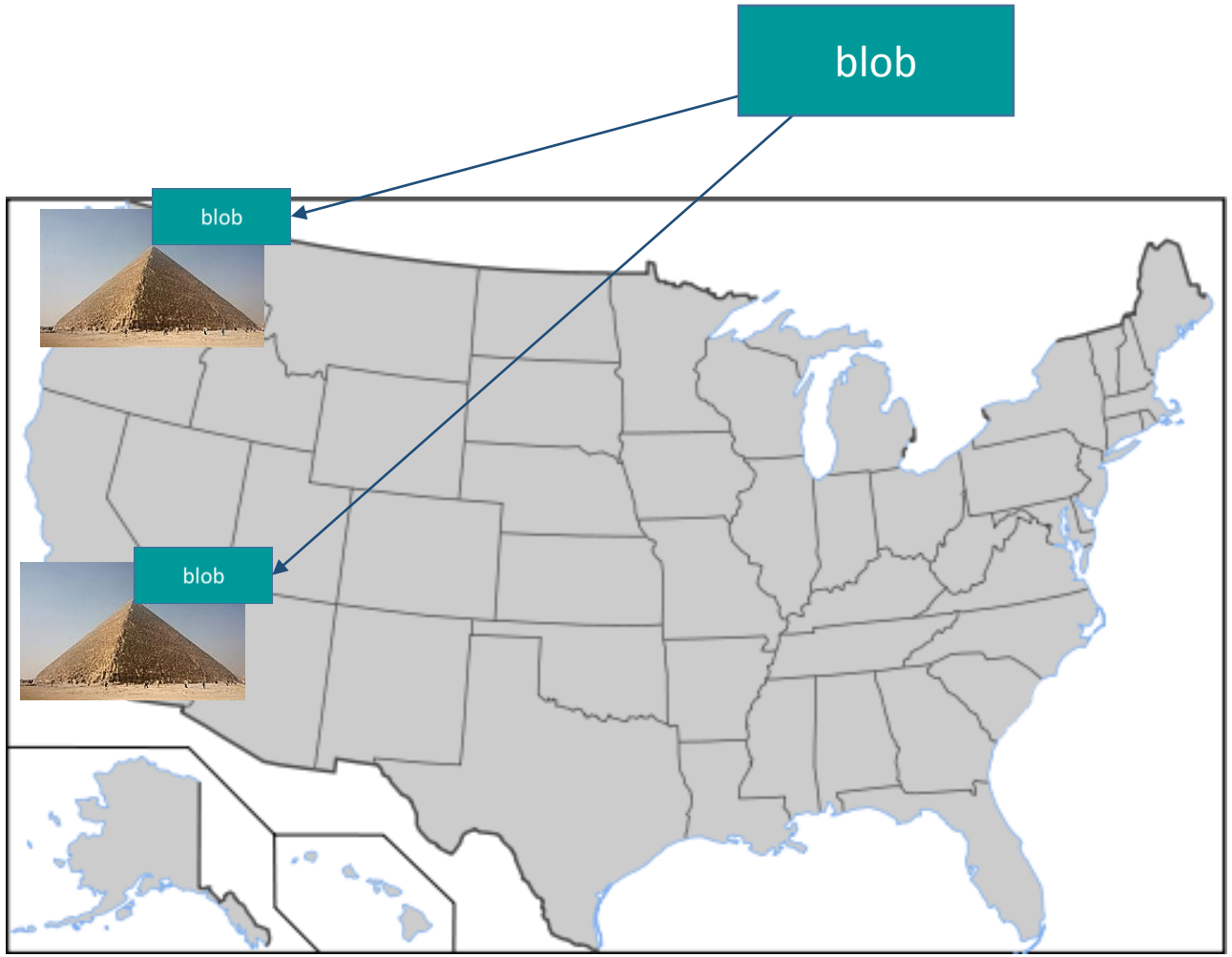
 New York University, *Microsoft Corporation

USENIX ATC, Santa Clara, CA, July 2017

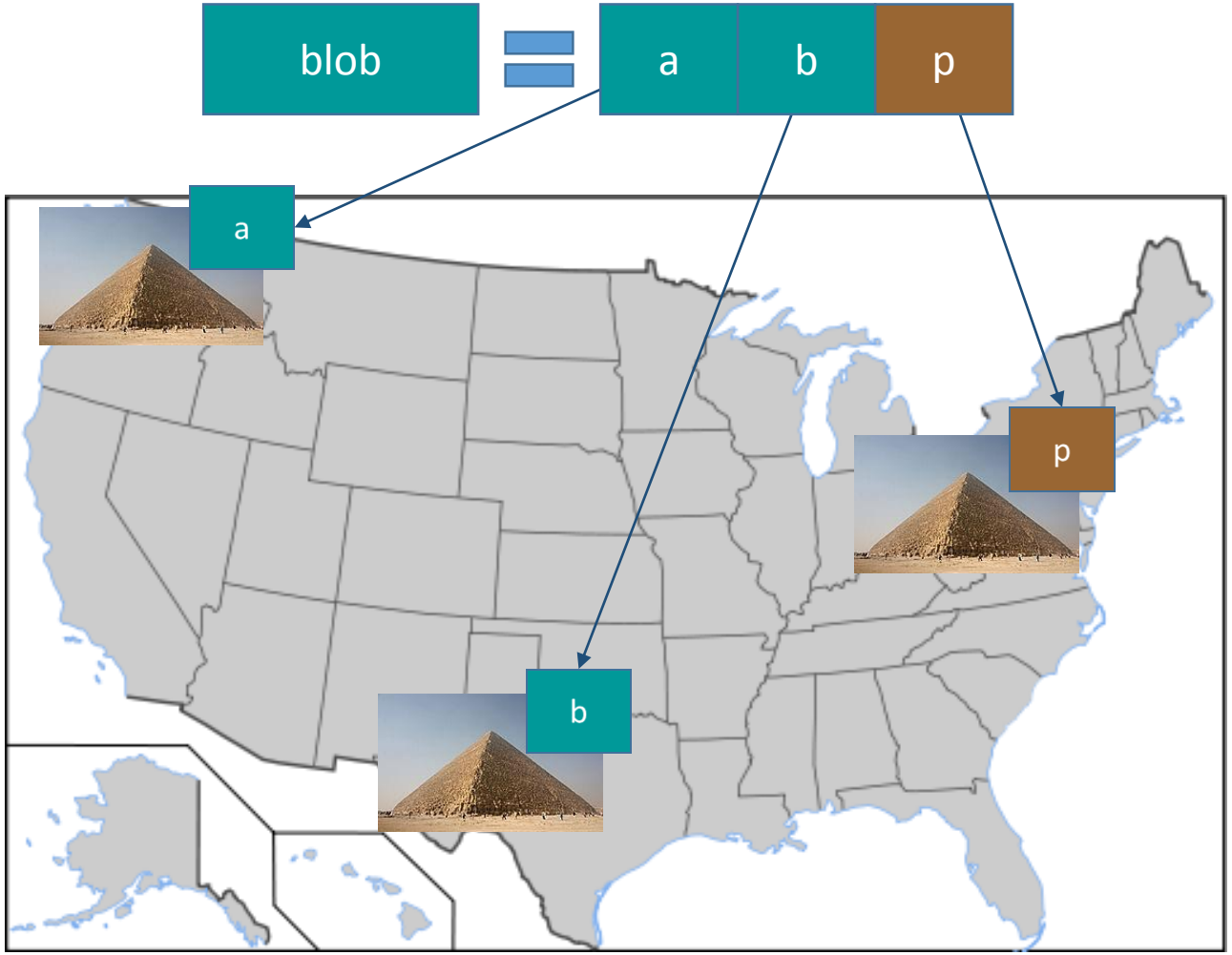
Azure data centers span the globe



Data center fault tolerance: Geo Replication

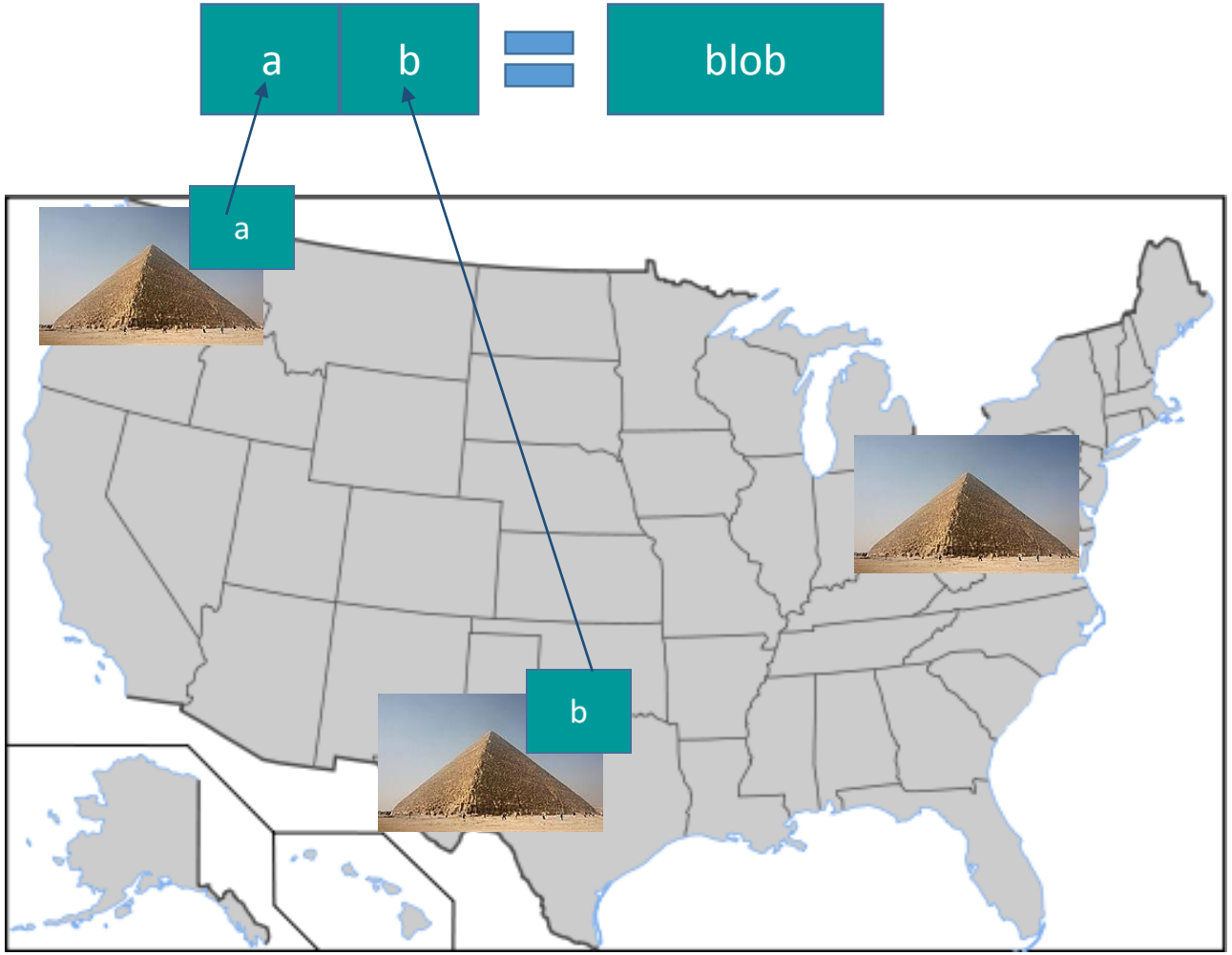


Data center fault tolerance: Erasure coding



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.

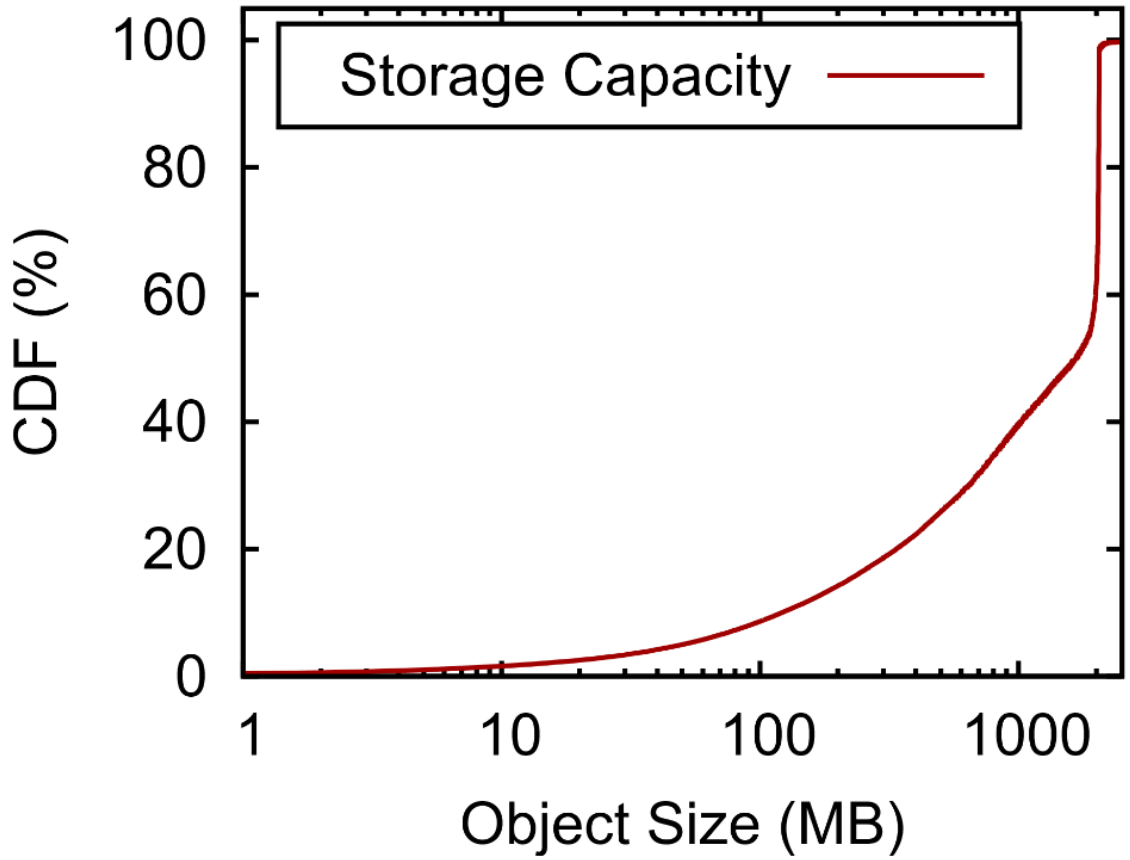
Cross DC communication is becoming less expensive

Largest bandwidth capacity today with 160Tpbs

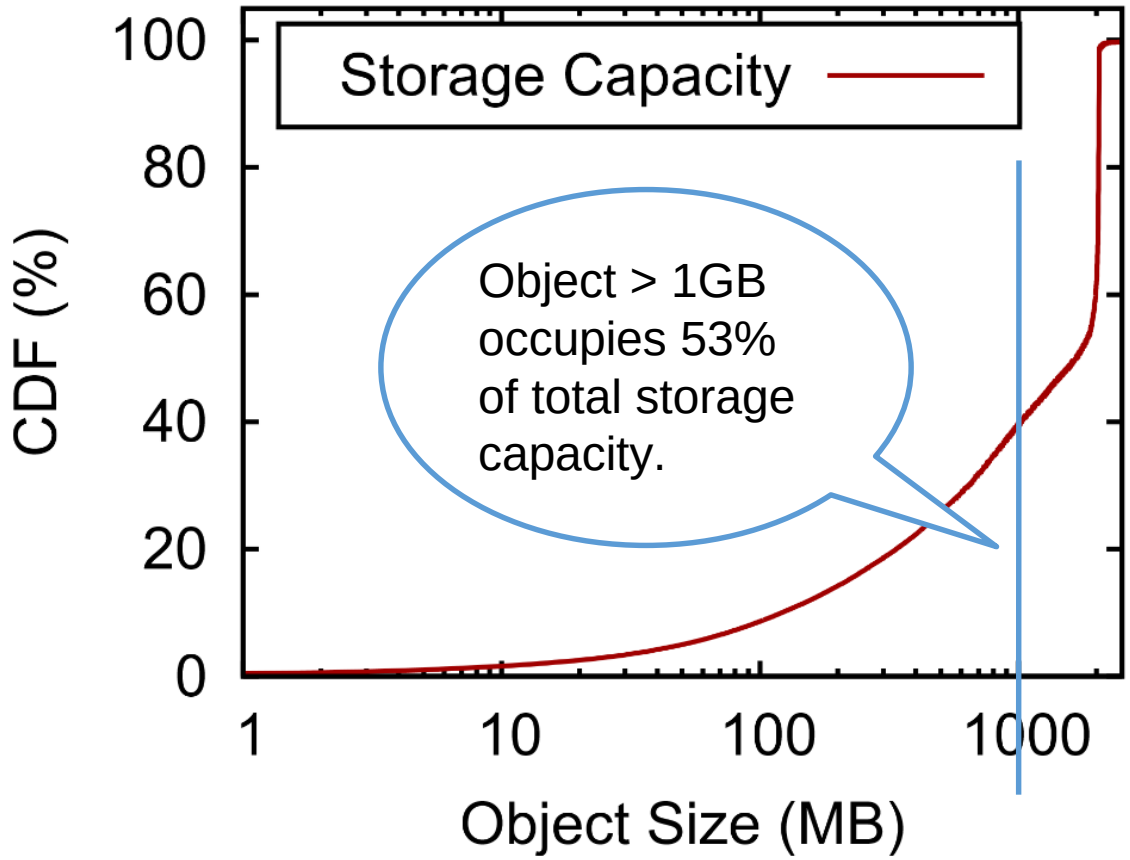
In search of a coding friendly workload

3 months trace: January – March 2016

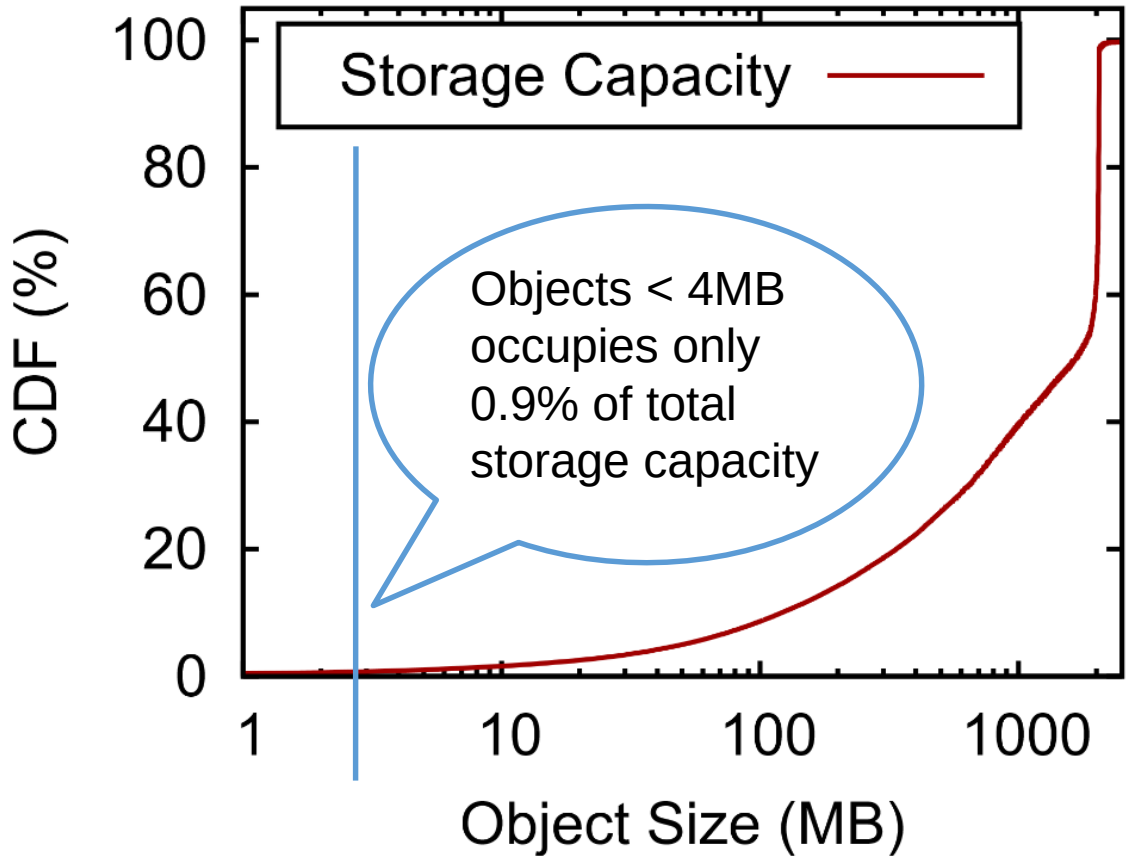
OneDrive is occupied mostly by large objects



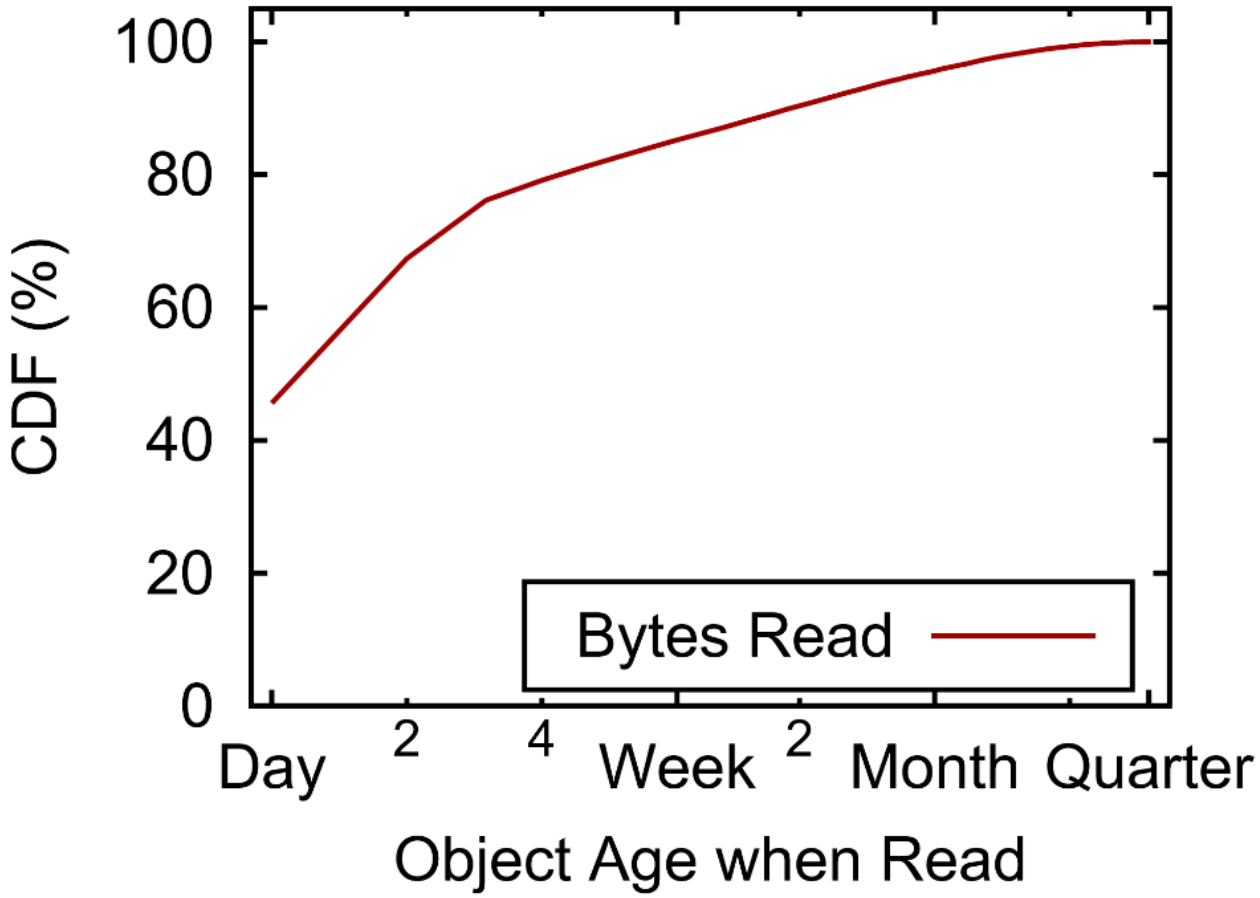
OneDrive is occupied mostly by large objects



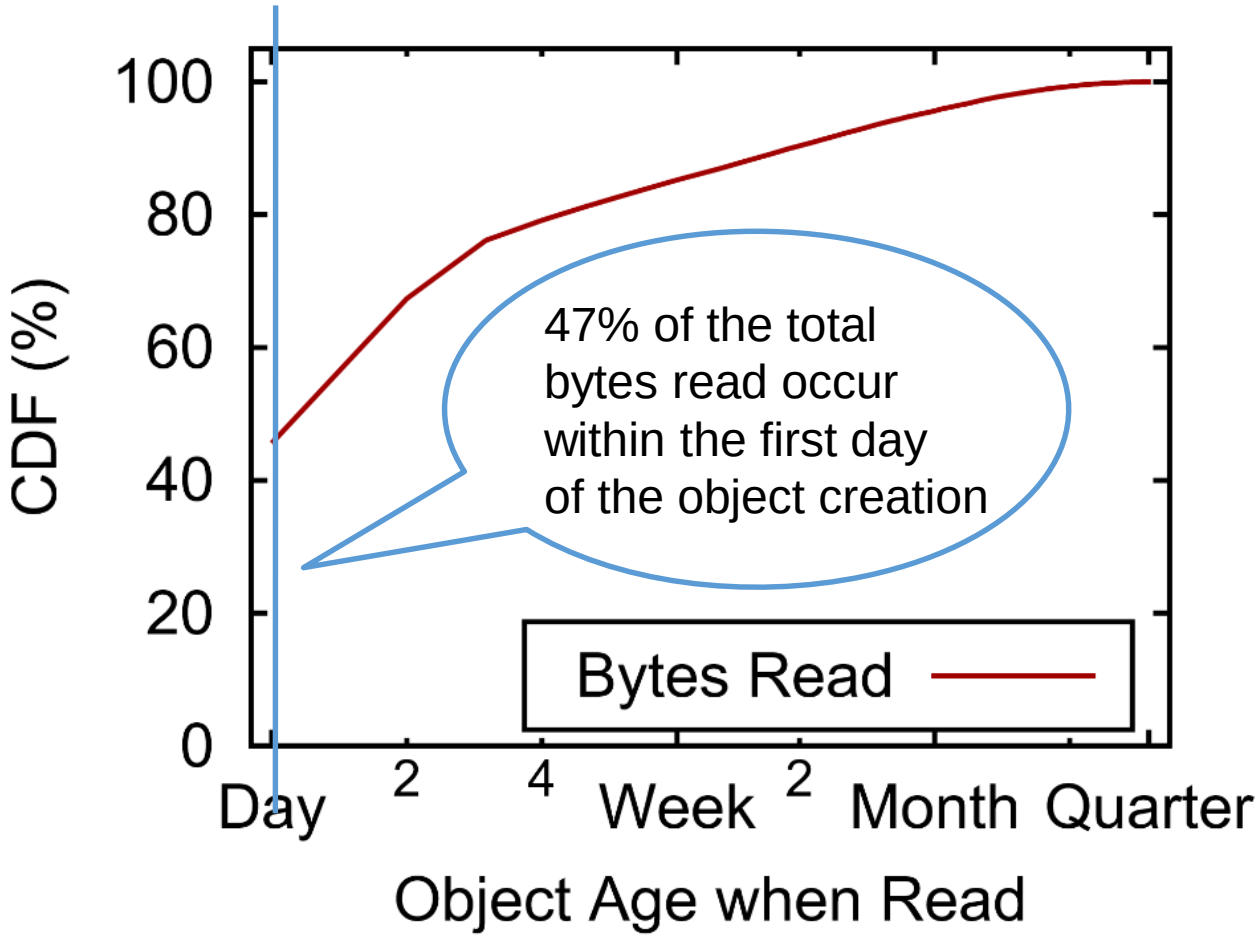
OneDrive is occupied mostly by large objects



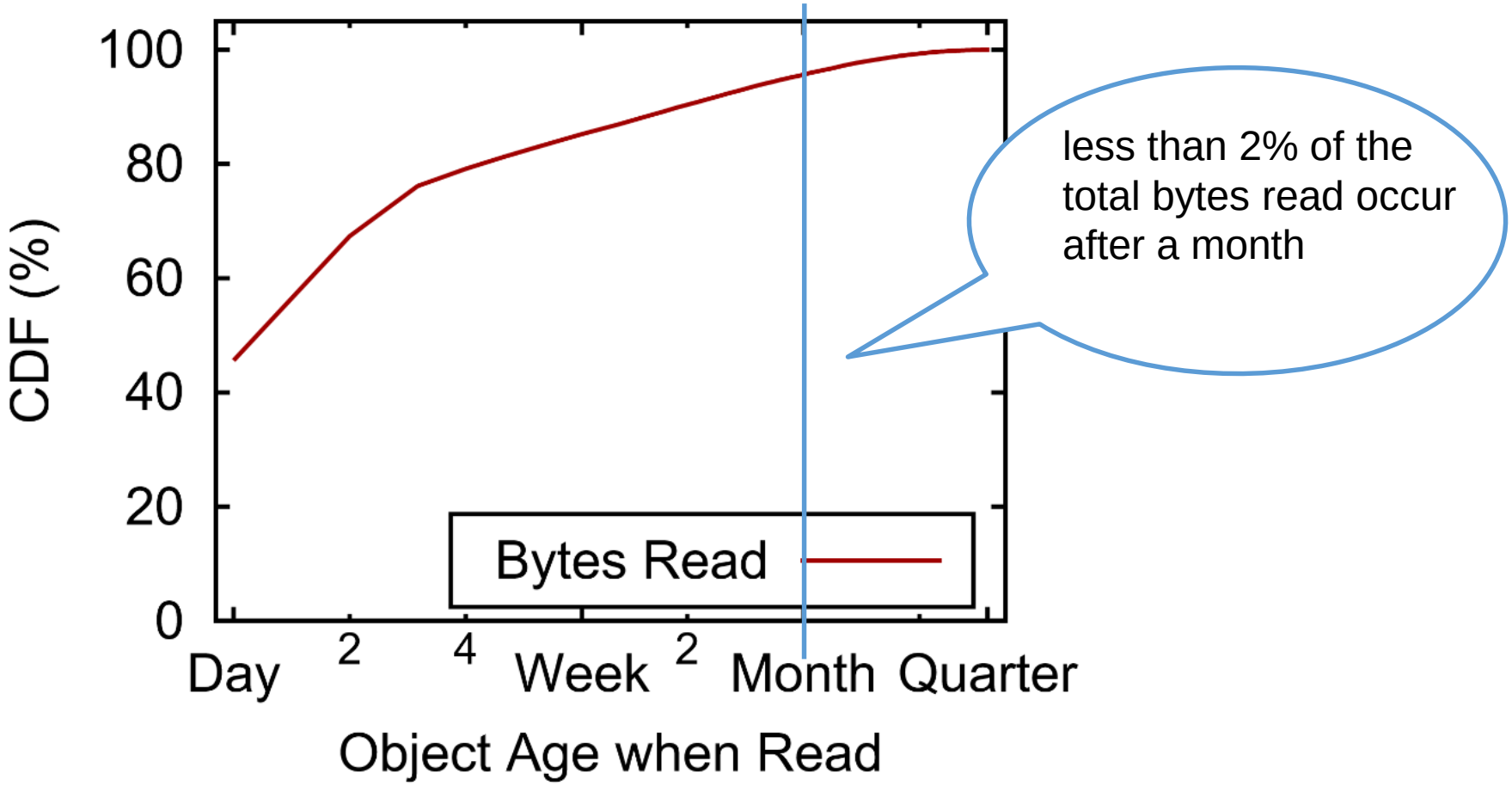
OneDrive reads are infrequent after a month



OneDrive reads are infrequent after a month



OneDrive reads are infrequent after a month



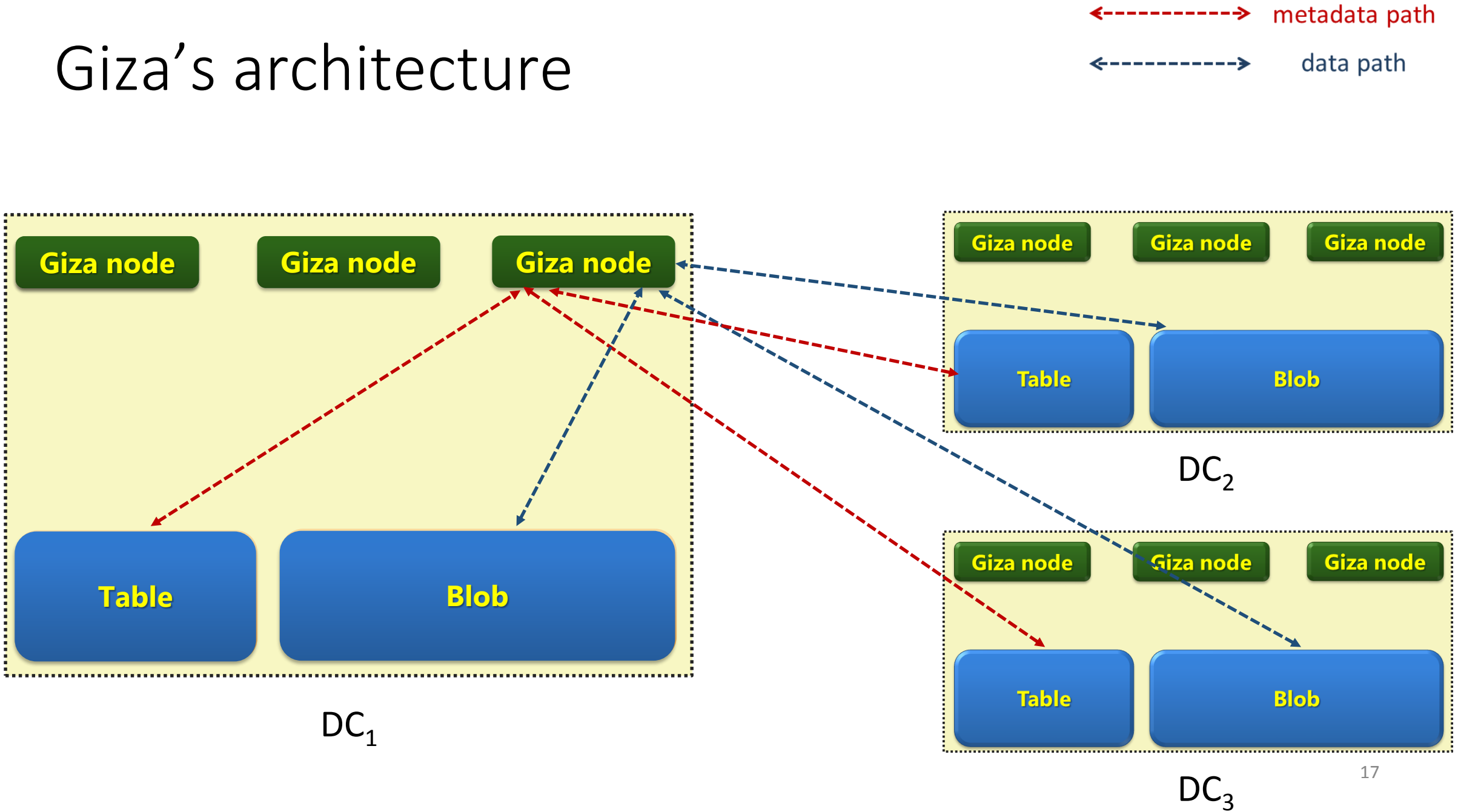
Outline

1. The case for erasure coding across data centers.
2. Giza design and challenges
3. Experimental Results

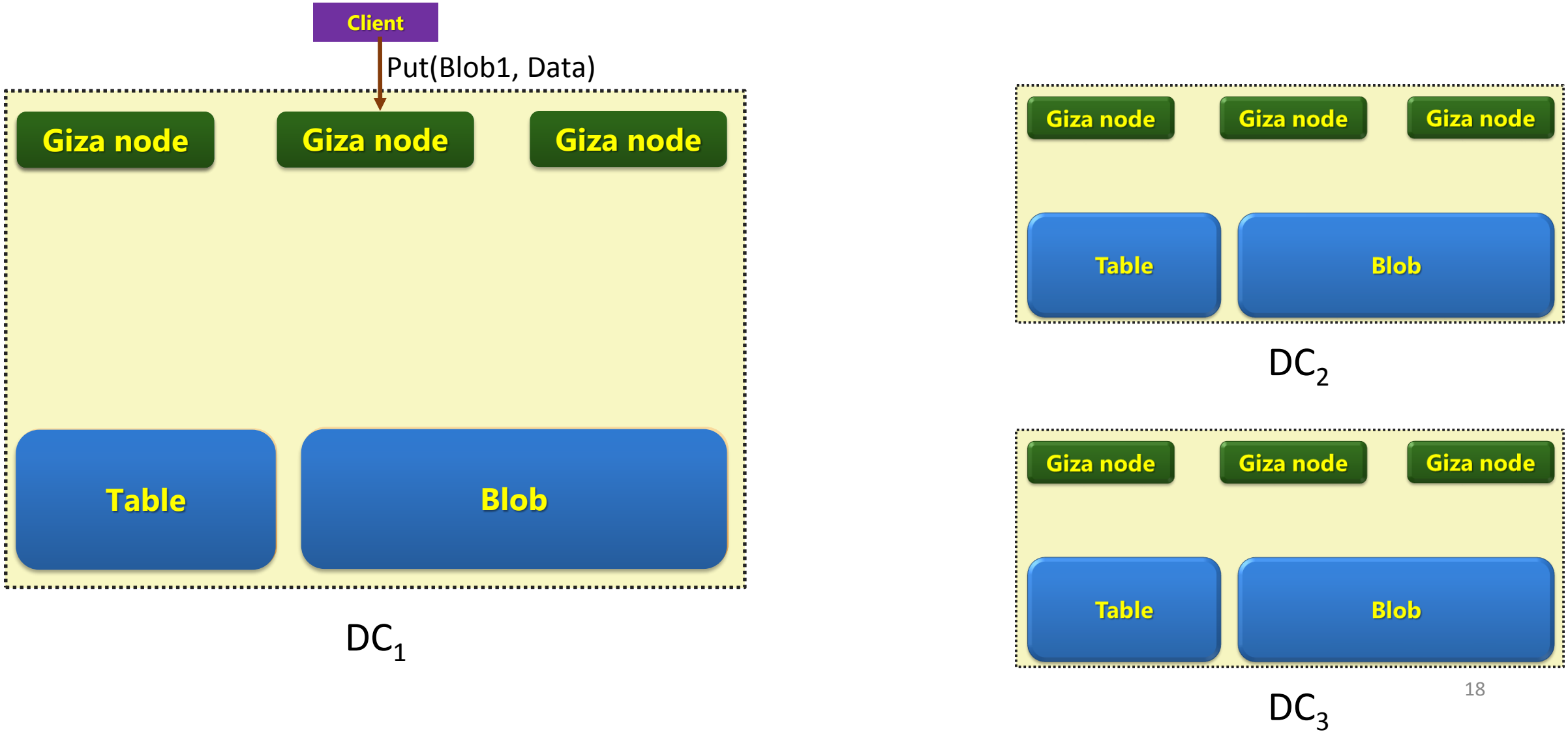
Giza: a strongly consistent versioned object store

- Erasure code each object individually.
- Optimize latency for Put and Get.
- Leverage existing single-DC cloud storage API.

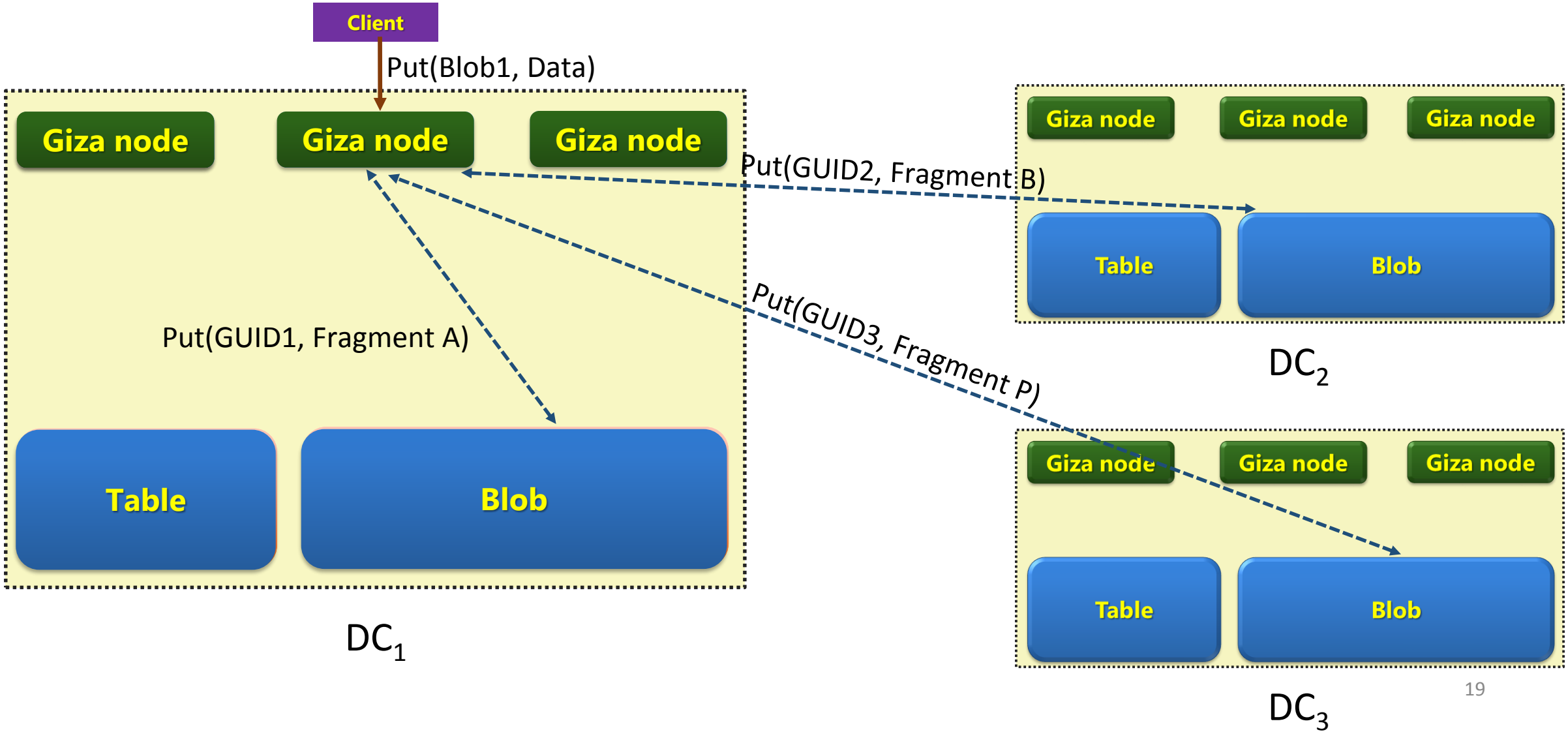
Giza's architecture



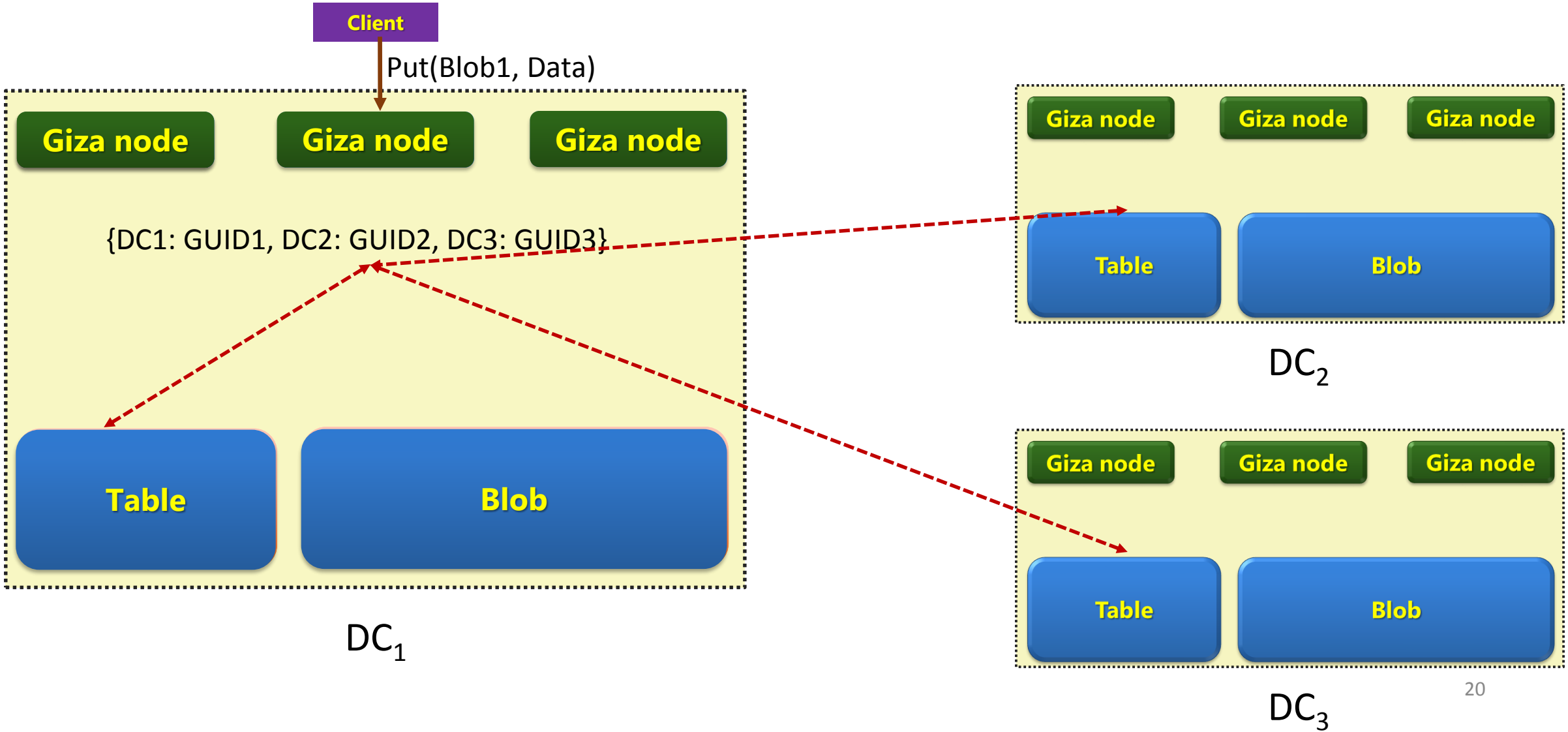
Giza's workflow



Giza's workflow



Giza's workflow



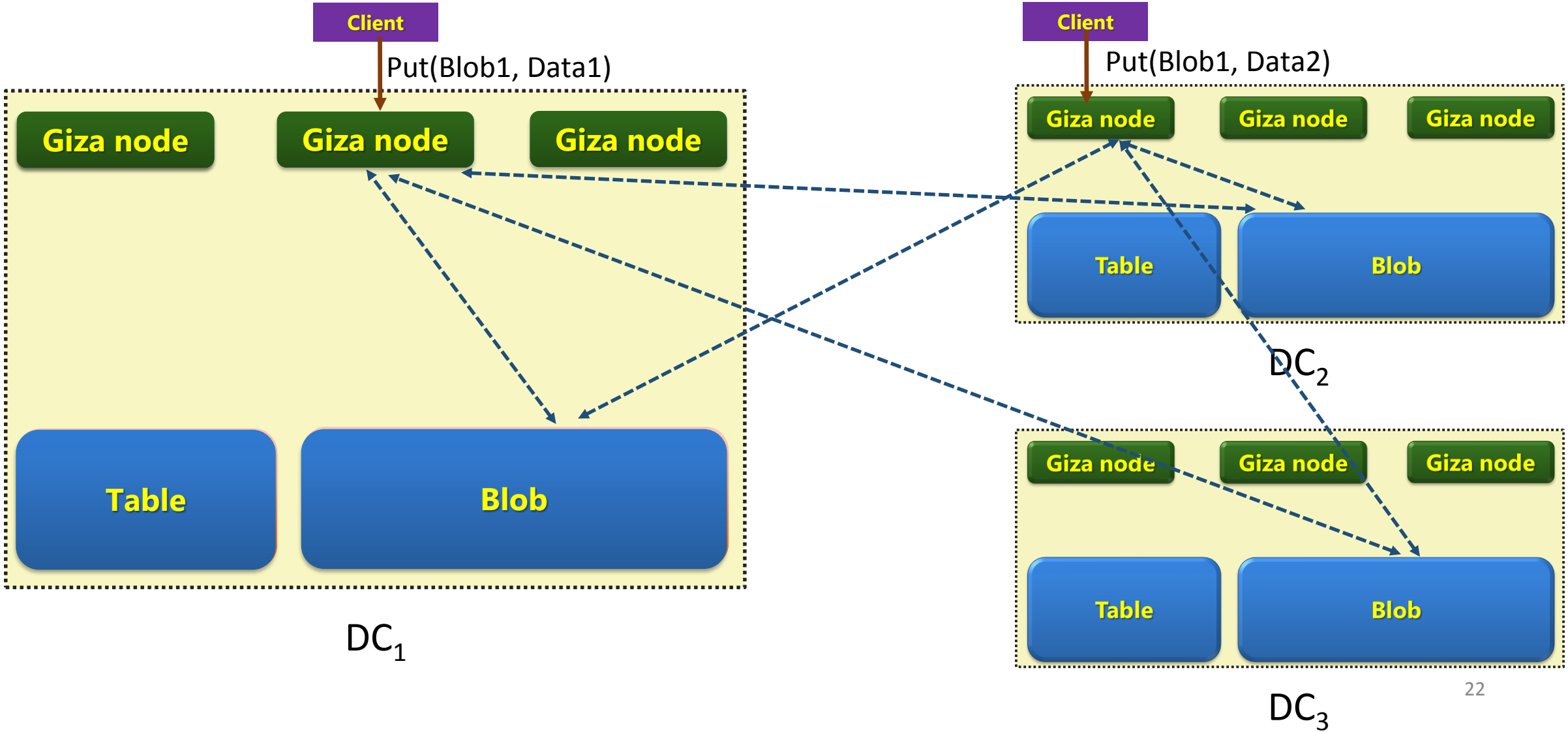
Giza metadata versioning

Let $md_{this} = \{DC_1{:}GUID_4, DC_2{:}GUID_5, DC_3{:}GUID_6\}$

Key	Metadata (version #: actual metadata)
Blob1	v1: md_1

Key	Metadata (version #: actual metadata)
Blob1	v1: md_1, v2: md_{this}

Inconsistency during concurrent put operations



Inconsistency during concurrent put operations

Order of arrival

md2

md1

Key	Metadata
Blob1	v1: md_1

DC_1

md1

md2

Key	Metadata
Blob1	v1: md_2

DC_2

md2

md1

Key	Metadata
Blob1	v1: md_1

DC_3

WARNING: INCONSISTENT STATE

Inconsistency during concurrent put operations

Order of arrival

md2

md1

Key	Metadata
Blob1	v1: md_1

DC_1

md1

md2

Key	Metadata
Blob1	v1: md_2

DC_2

md2

md1

Key	Metadata
Blob1	v1: md_1

FAILURE!

DC_3

WARNING: INCONSISTENT STATE

Choosing version value via consensus

Key	Metadata
Blob1	v1: md_1 v2: md_2

DC_1

Key	Metadata
Blob1	v1: md_1 v2: md_2

DC_2

Key	Metadata
Blob1	v1: md_1 v2: md_2

DC_3

OR

Key	Metadata
Blob1	v1: md_2 v2: md_1

DC_1

Key	Metadata
Blob1	v1: md_2 v2: md_1

DC_2

Key	Metadata
Blob1	v1: md_2 v2: md_1

DC_3

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

Paxos

- Can commit with 2 round trips.
- Requires majority of replicas to commit.

  $\frac{N}{2} + 1$

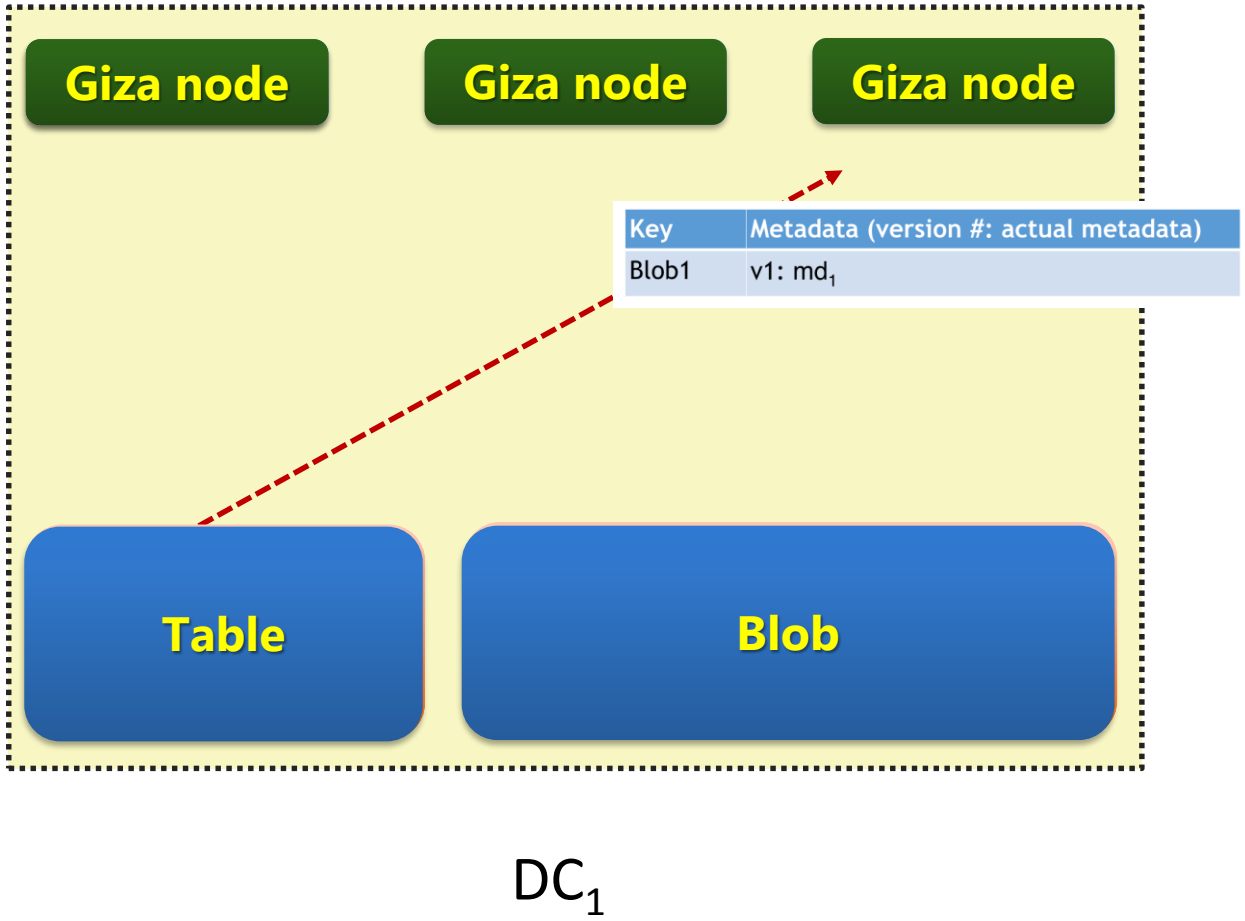
Fast Paxos

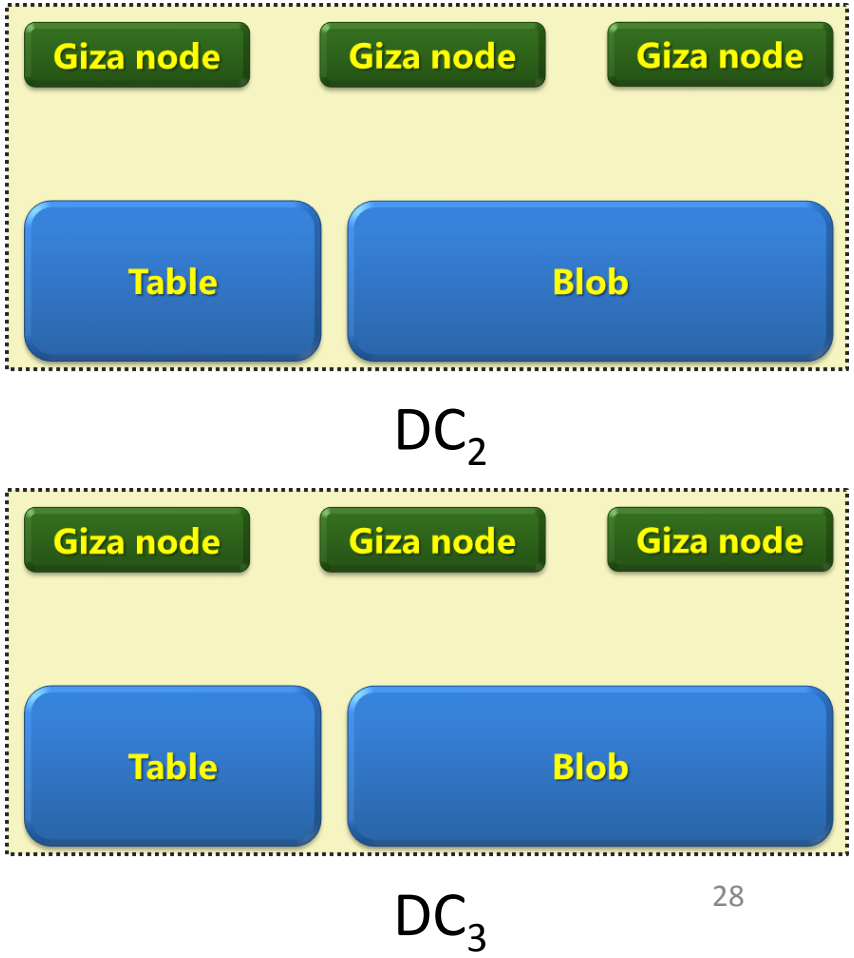
- Can commit with 1 round trip.
- Requires more than the majority of replicas to commit.

  $\frac{2N}{3} + 1$

Implementing Fast Paxos with Azure table API

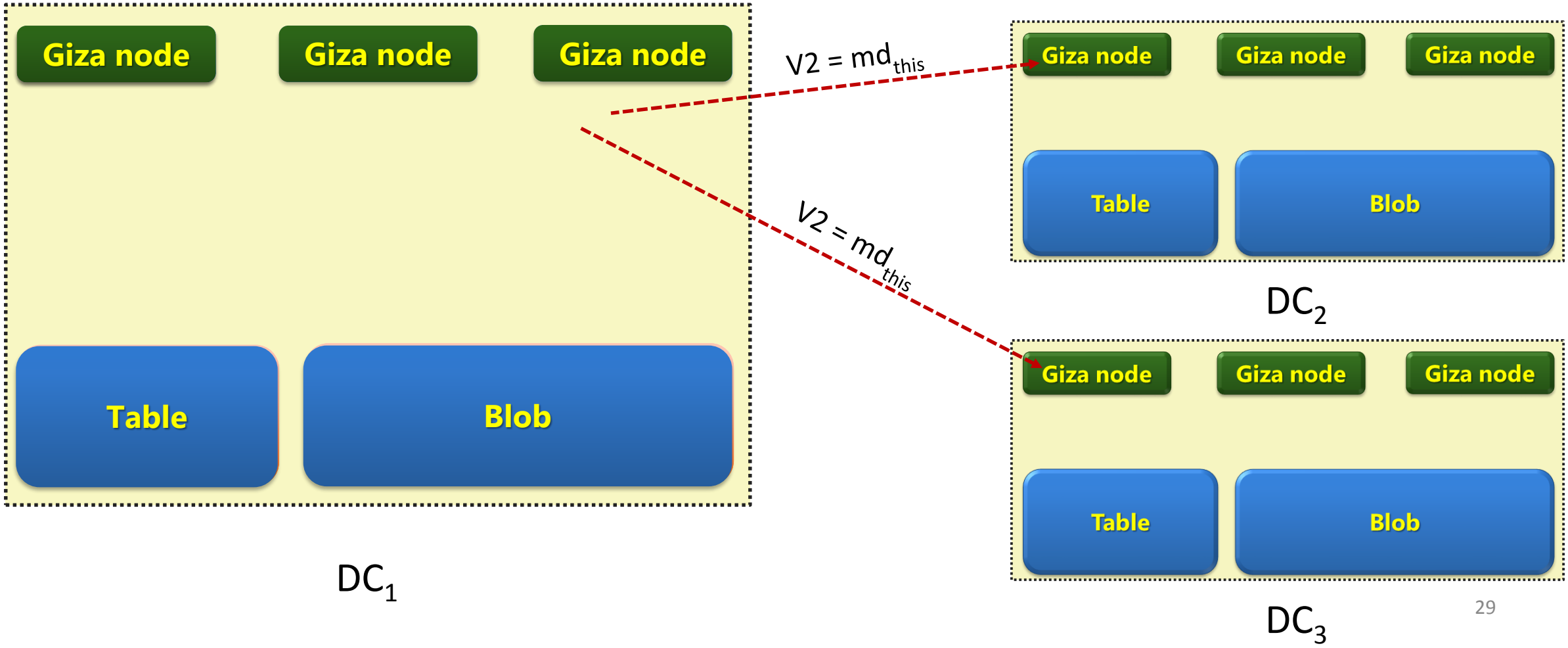
Let md_{this} = ({DC_1:$GUID_1$, DC_2:$GUID_2$, DC_3:$GUID_3$}, Paxos states)





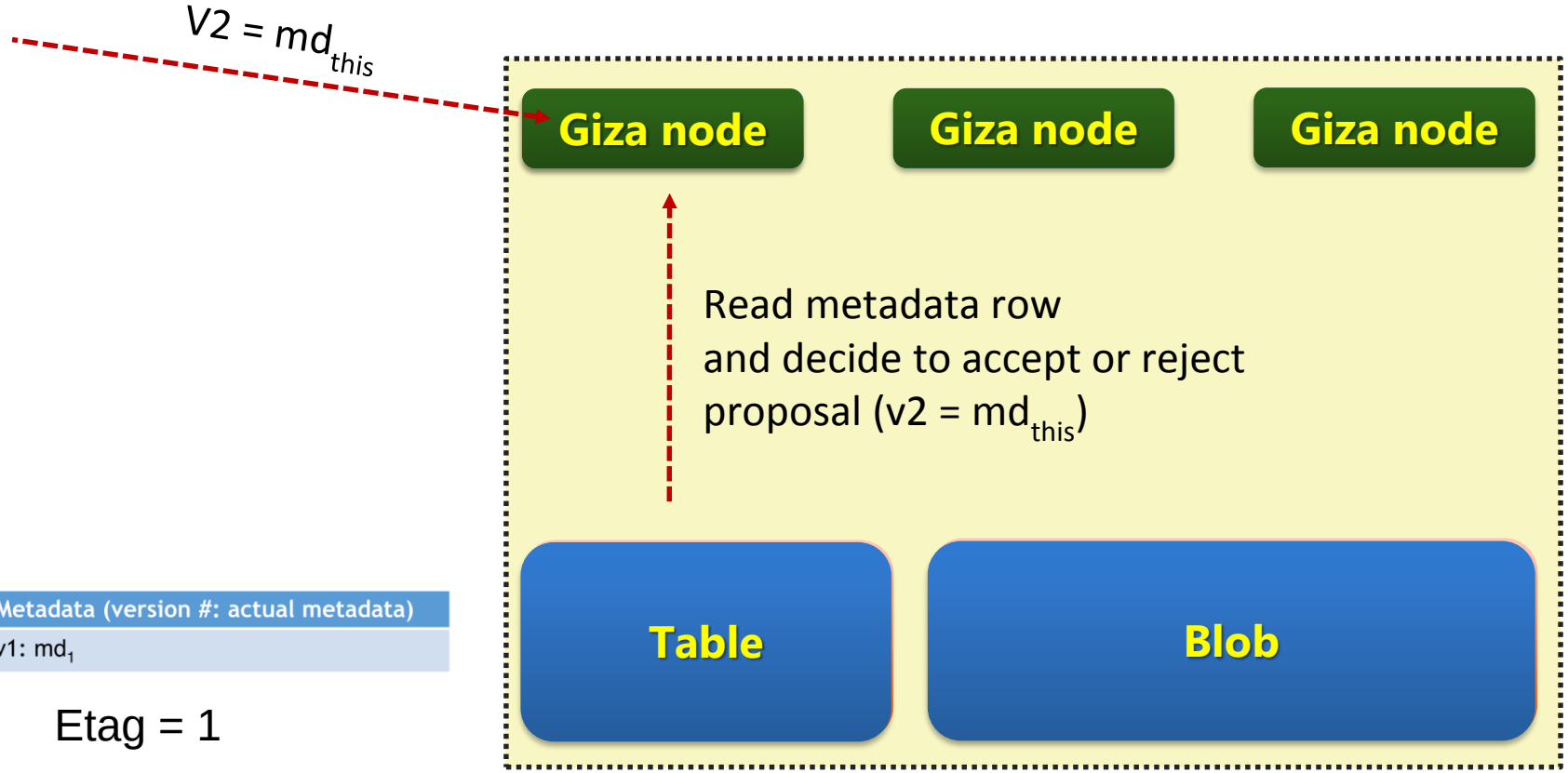
Implementing Fast Paxos with Azure table API

Let md_{this} = ({DC_1:$GUID_1$, DC_2:$GUID_2$, DC_3:$GUID_3$}, 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)



Key	Metadata (version #: actual metadata)
Blob1	v1: md_1

Etag = 1

Remote DC

Implementing Fast Paxos with Azure table API

Let md_{this} = ({DC_1:$GUID_1$, DC_2:$GUID_2$, DC_3:$GUID_3$}, Paxos states)

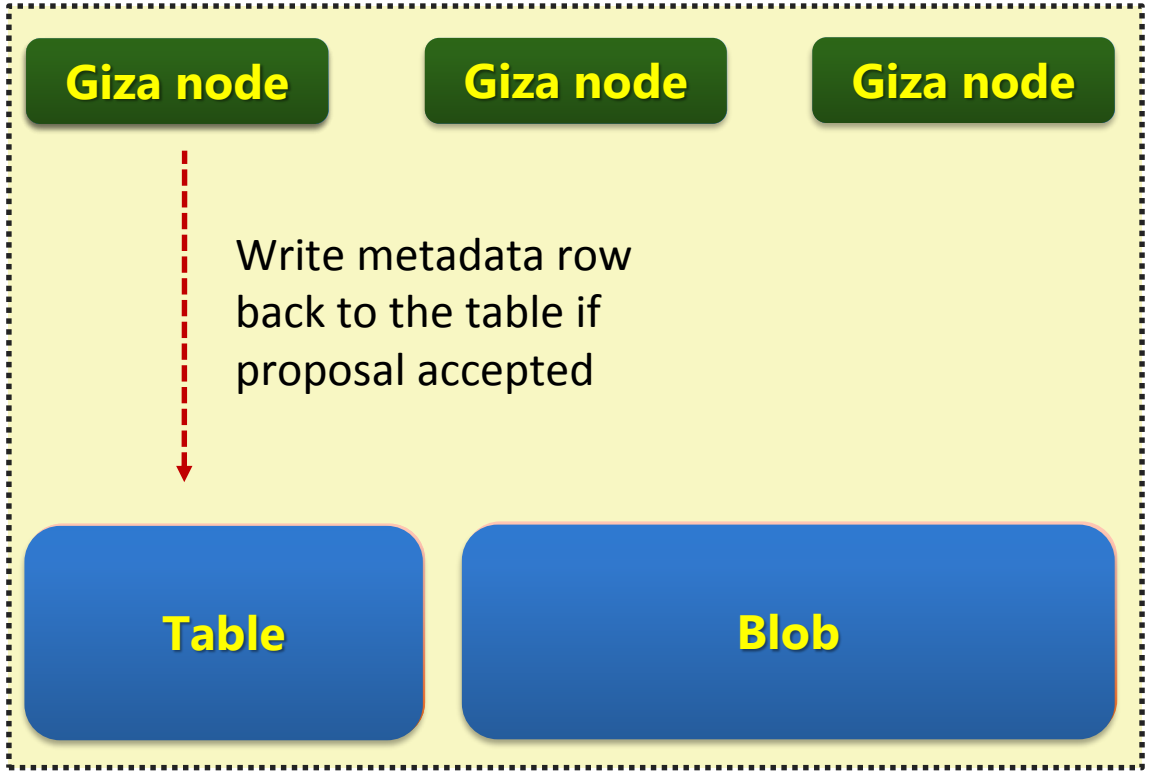
Etag = 1

Key	Metadata (version #: actual metadata)
Blob1	v1: md_1, v2: md_{this}

Fail to update table since etag has changed

Key	Metadata (version #: actual metadata)
Blob1	v1: md_1, v2: md_{that}

Etag = 2



Remote DC

Implementing Fast Paxos with Azure table API

Let md_{this} = ({DC_1:$GUID_1$, DC_2:$GUID_2$, DC_3:$GUID_3$}, Paxos states)

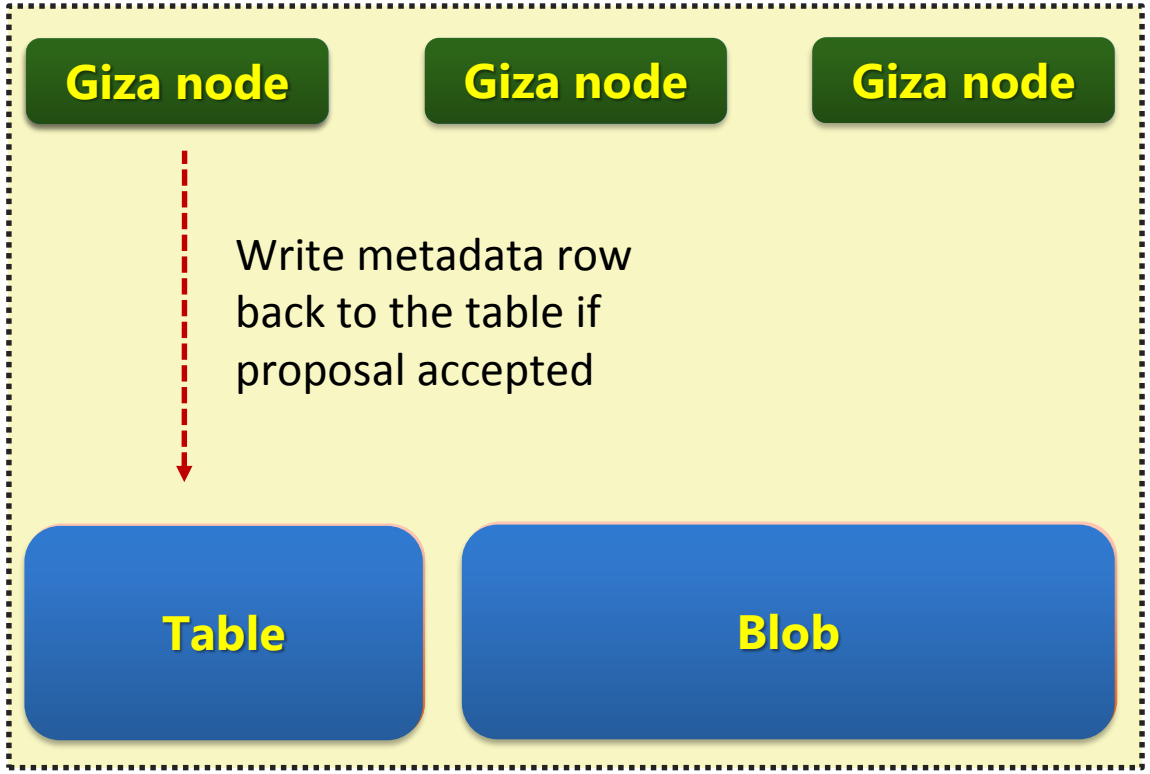
Etag = 1

Key	Metadata (version #: actual metadata)
Blob1	v1: md_1, v2: md_{this}

Successfully update table since etag hasn't changed

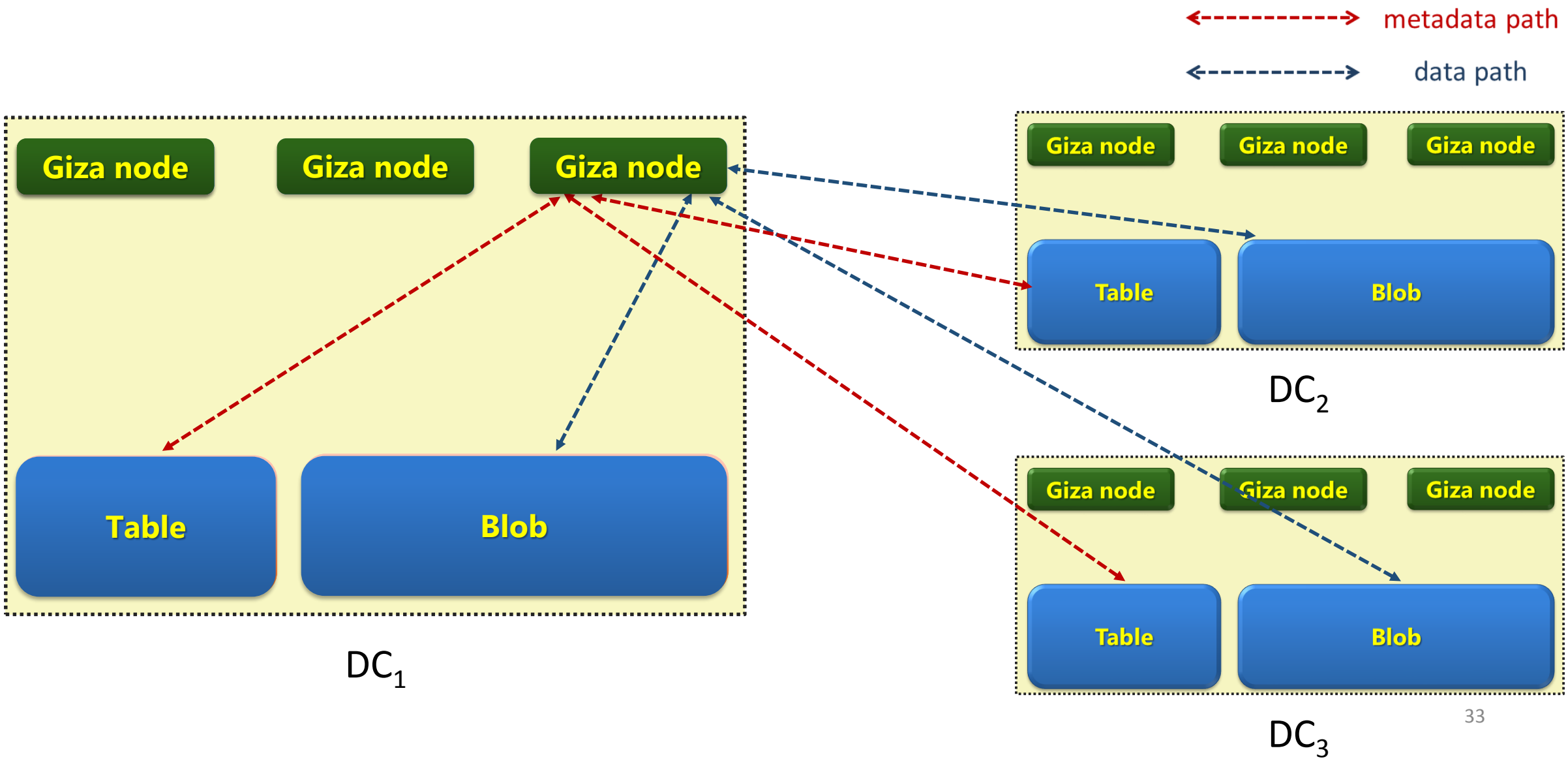
Key	Metadata (version #: actual metadata)
Blob1	v1: md_1

Etag = 1



Remote DC

Parallelizing metadata and data path



Data path fails, metadata path succeeds

Key	Metadata (version #: actual metadata)
Blob1	v1: md_1, v2: md_2

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.

Key	Committed	Metadata (version #: actual metadata)
Blob1	1	v1: md_1, v2: md_2

Ongoing, not committed

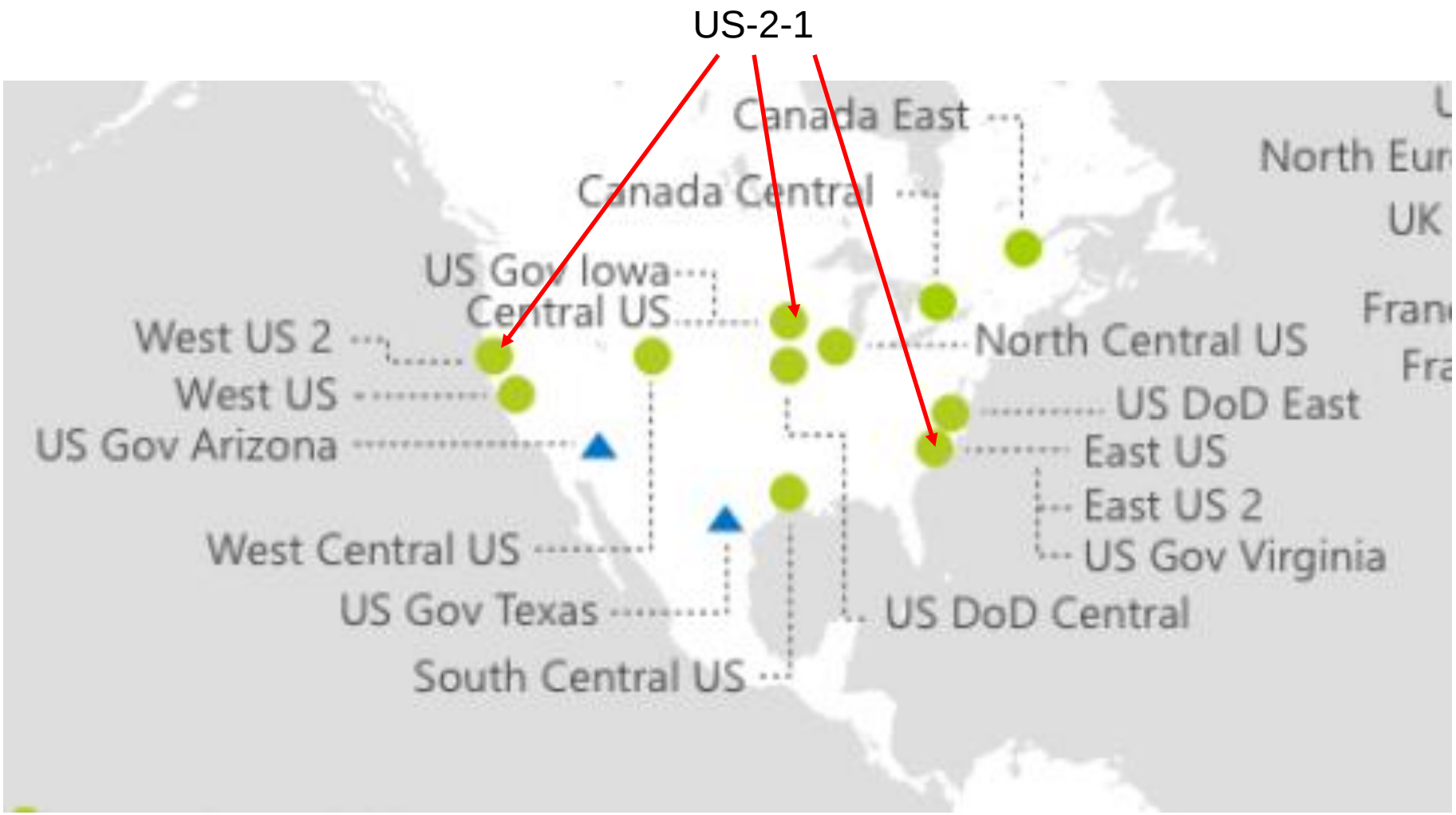
Giza Get

- 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.

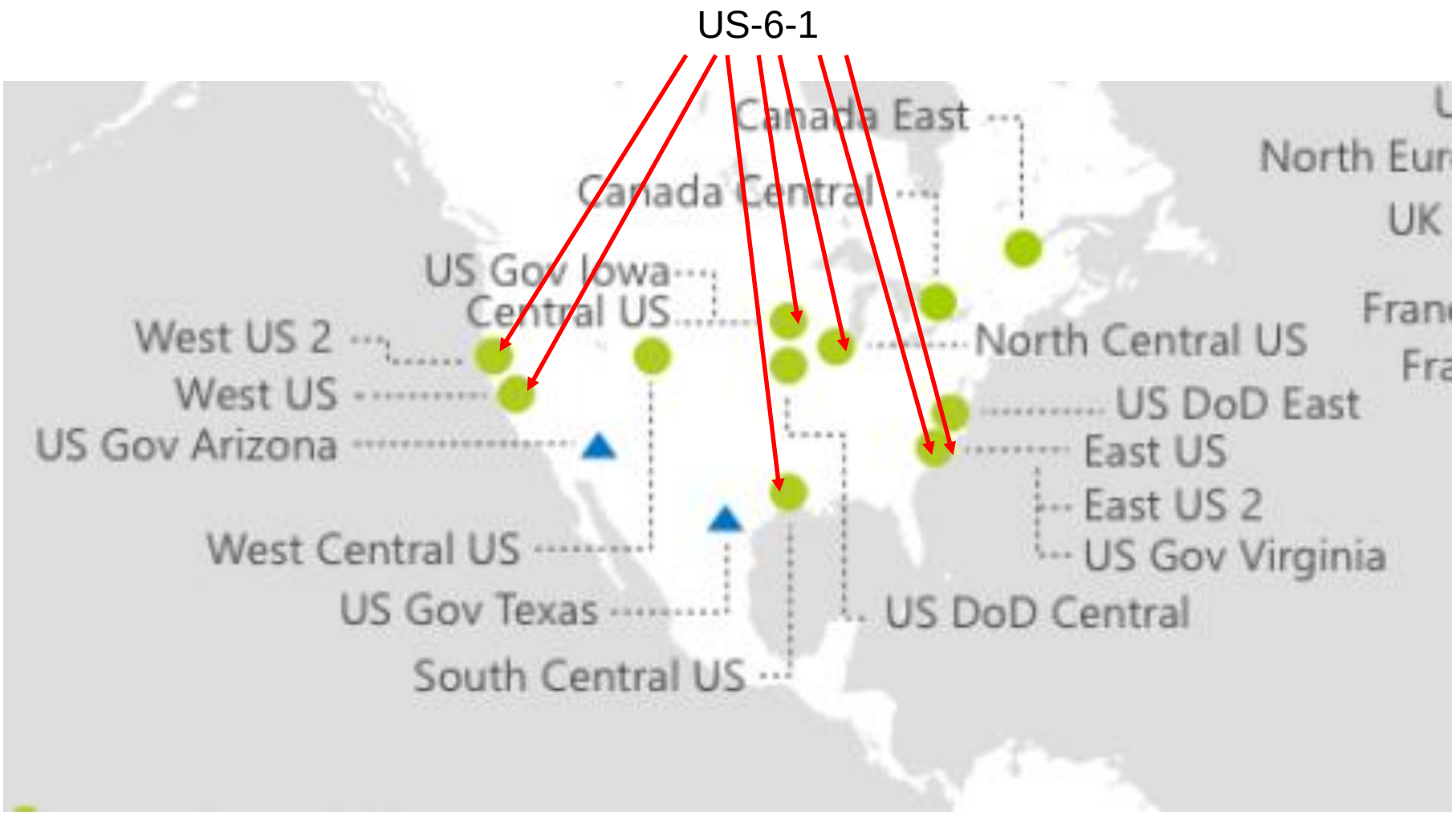
Outline

1. The case for erasure coding across data centers.
2. Giza design and challenges
3. Experimental Results

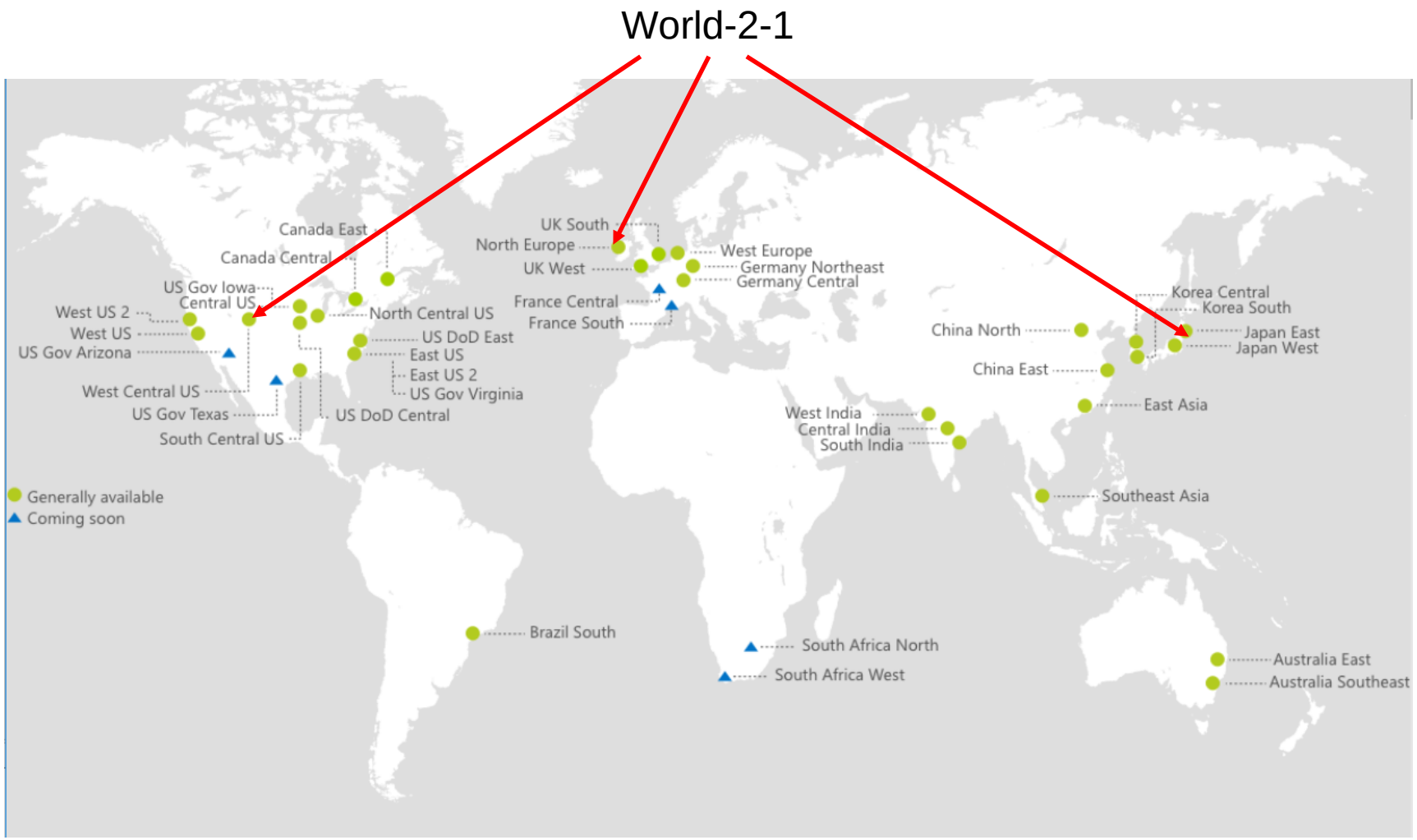
Giza deployment configurations



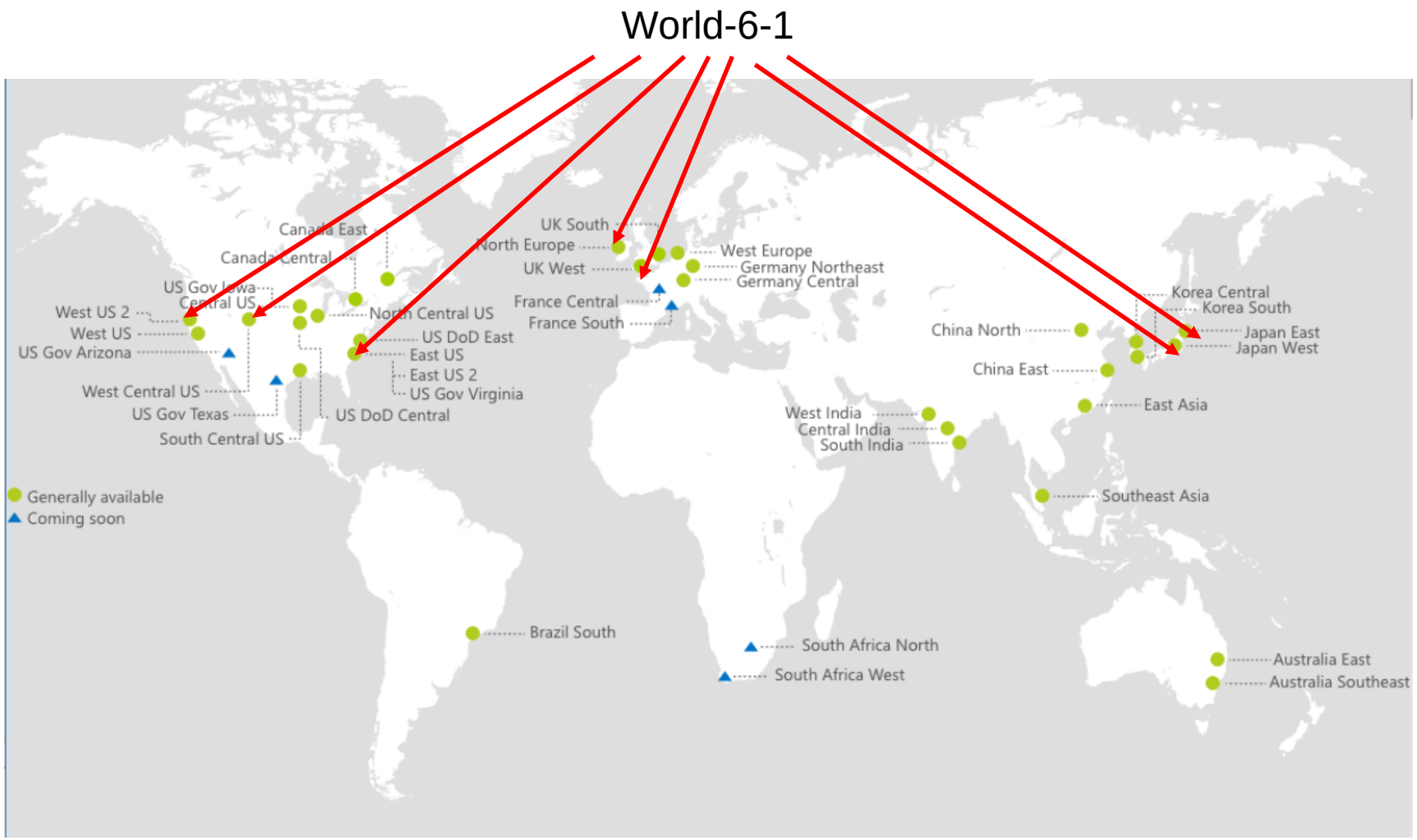
Giza deployment configurations



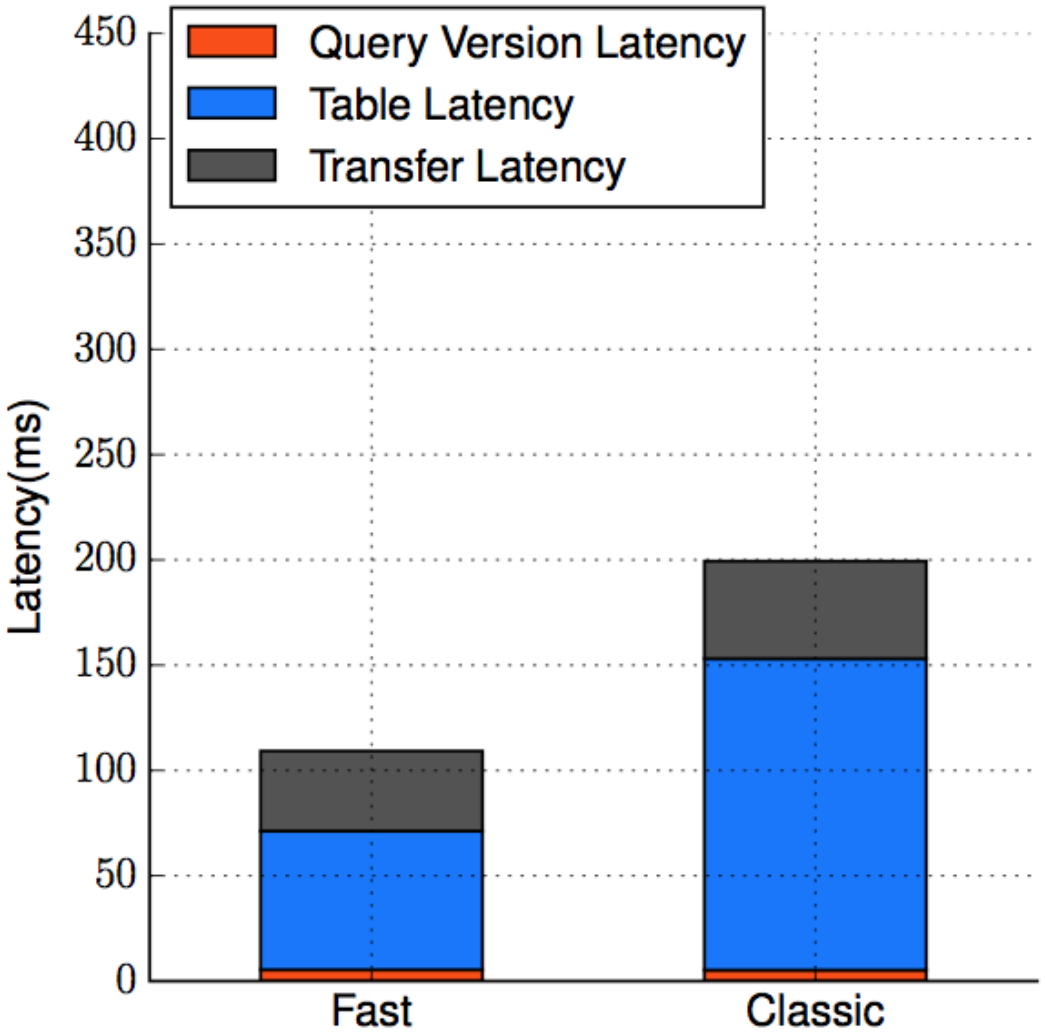
Giza deployment configurations

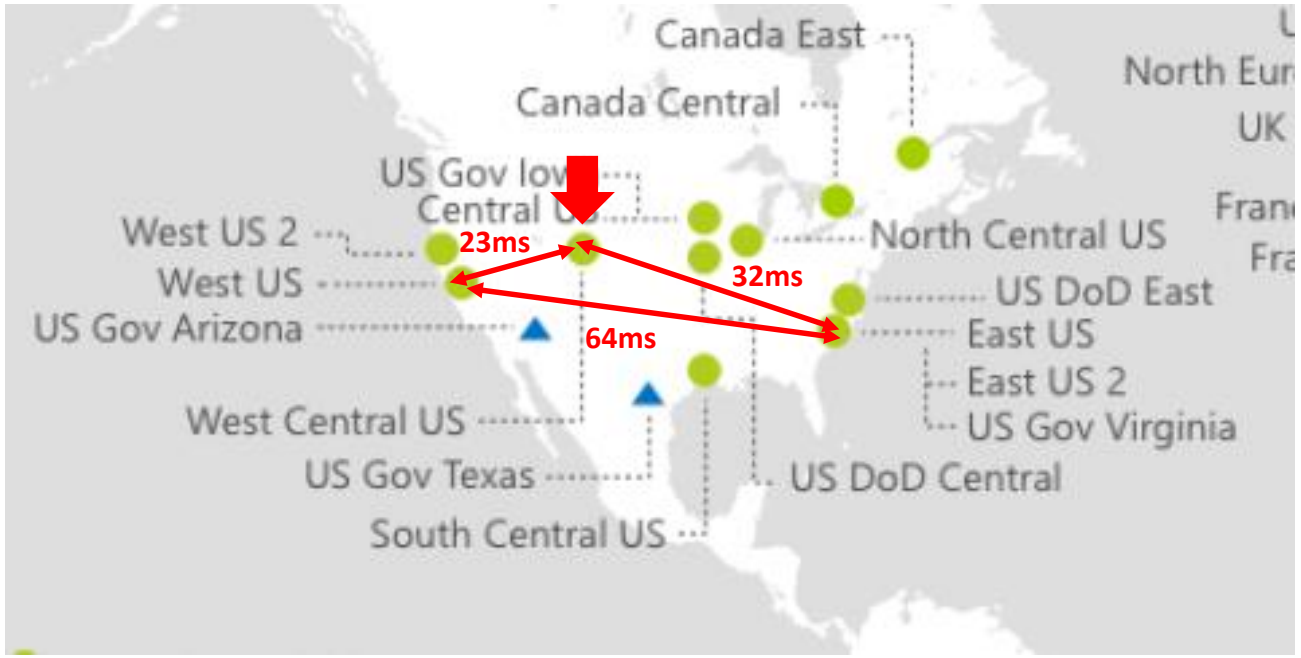


Giza deployment configurations

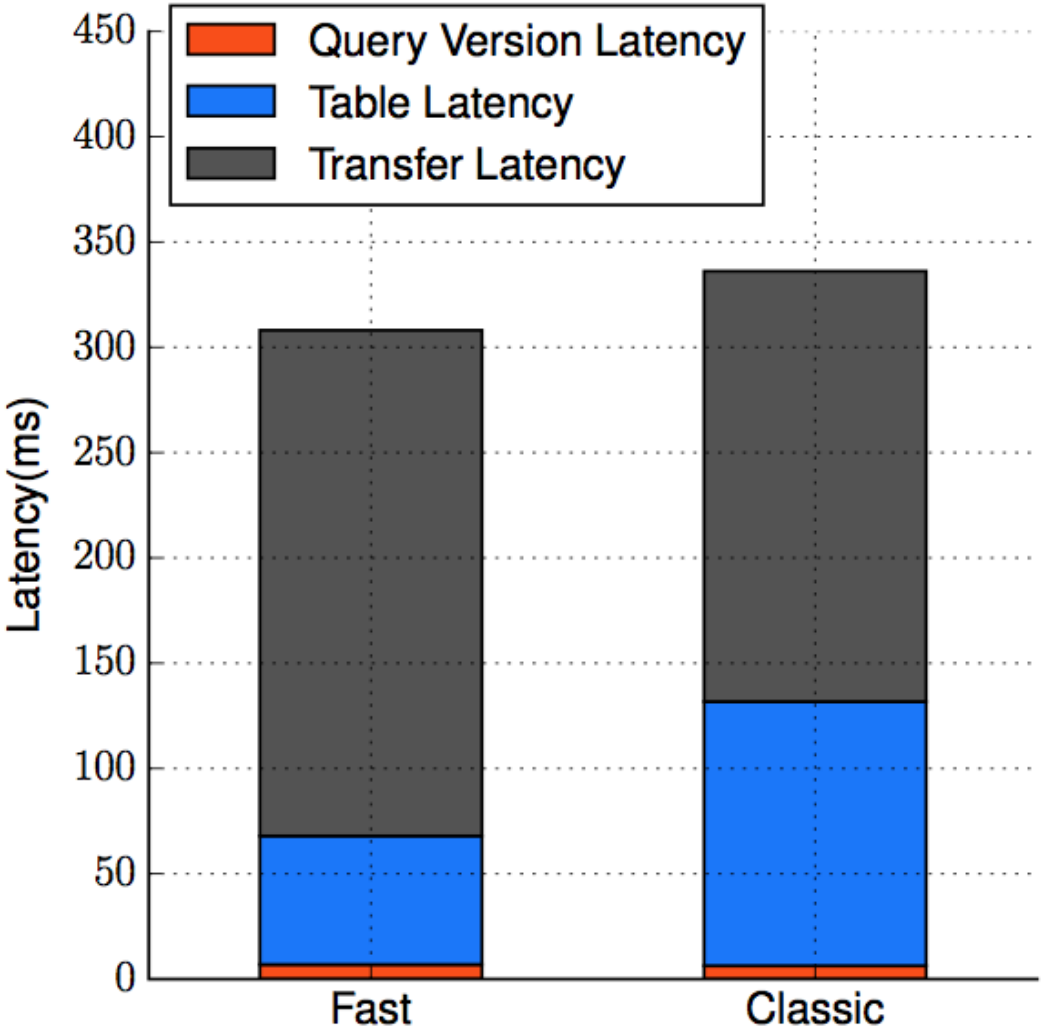


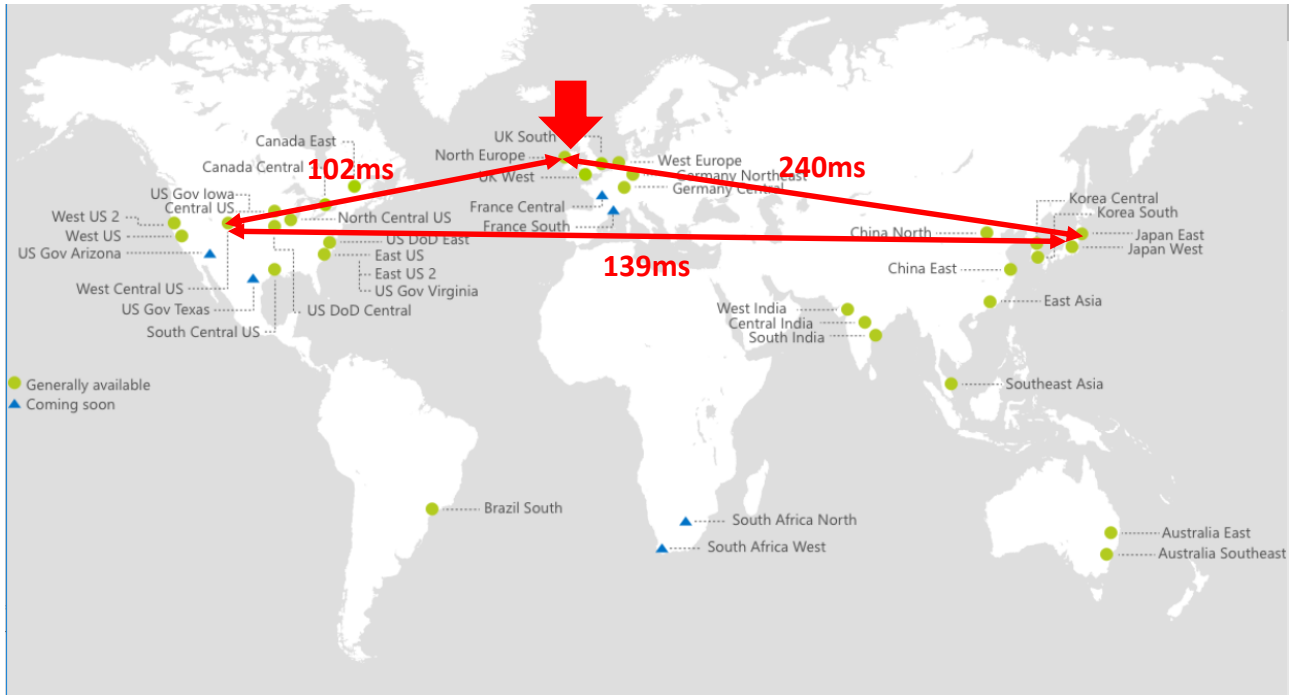
Metadata Path: Fast vs Classic Paxos



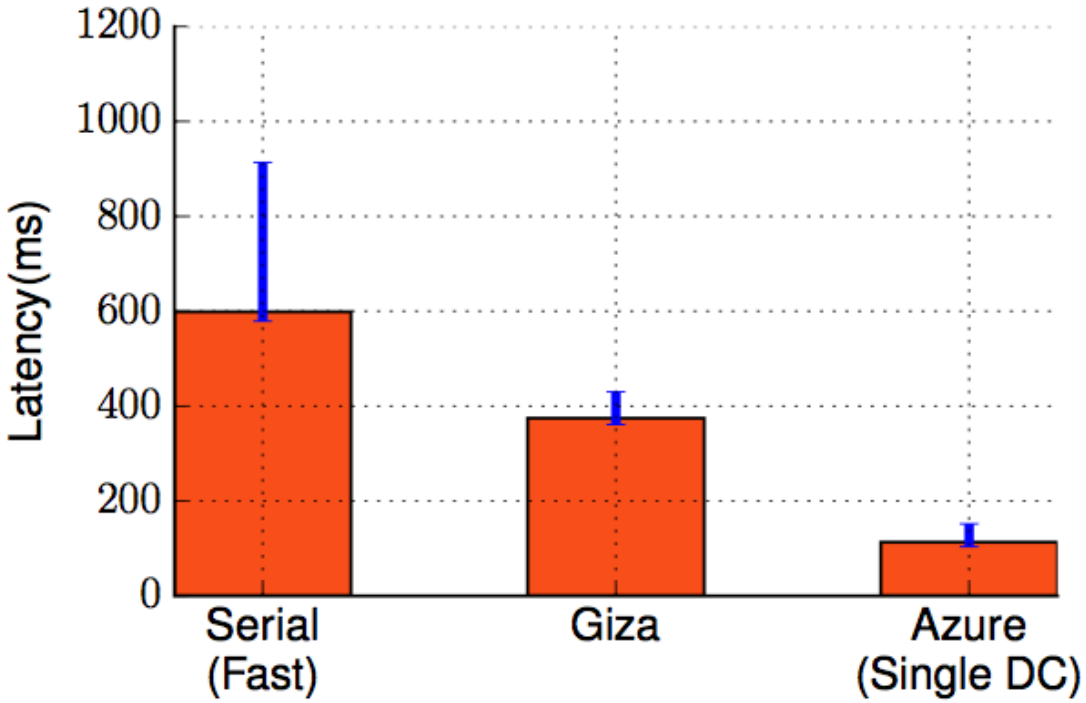


Metadata Path: Fast vs Classic Paxos

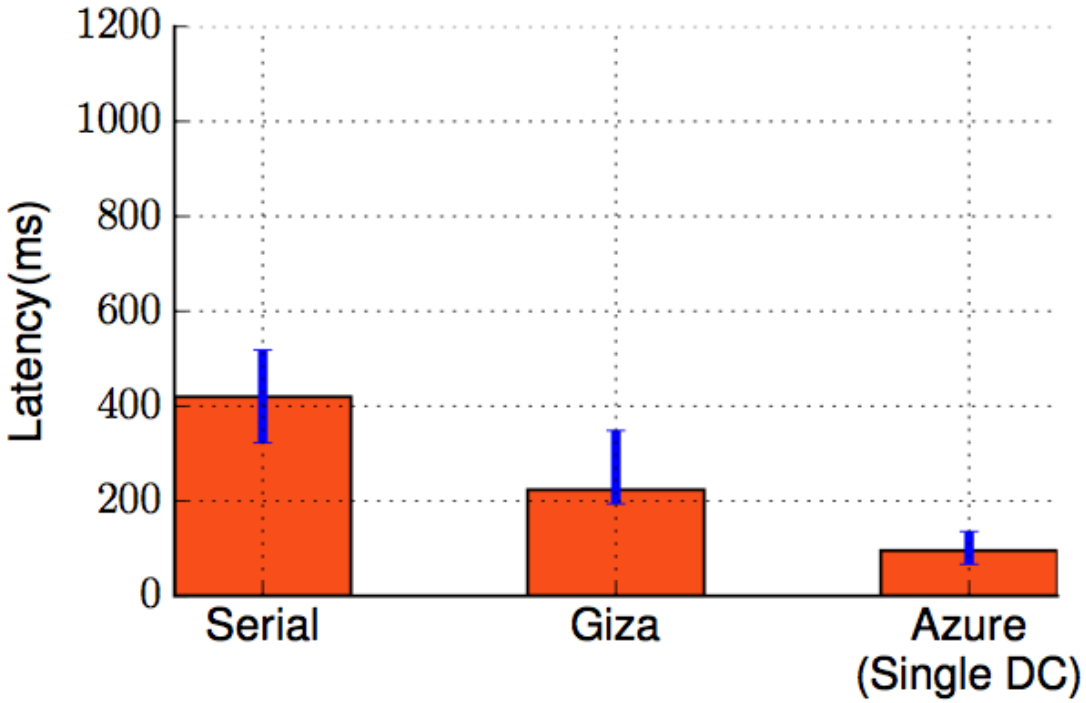




Giza Put Latency (World-2-1)



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.