Distributed Systems

15-440/640

Fall 2018

14 – Distributed Databases: Case Study

Readings: Spanner paper, Daniel Abadi's blog post

Announcements

Yuvraj's OH: 1pm to 2pm today

Daniel's OH: right after class to 1pm today

Piazza questions on Lamport and TO Multicast

→ will update lecture slides today

Next Tuesday: midterm review, Q&A

Next Thursday: midterm I, in class (4401)

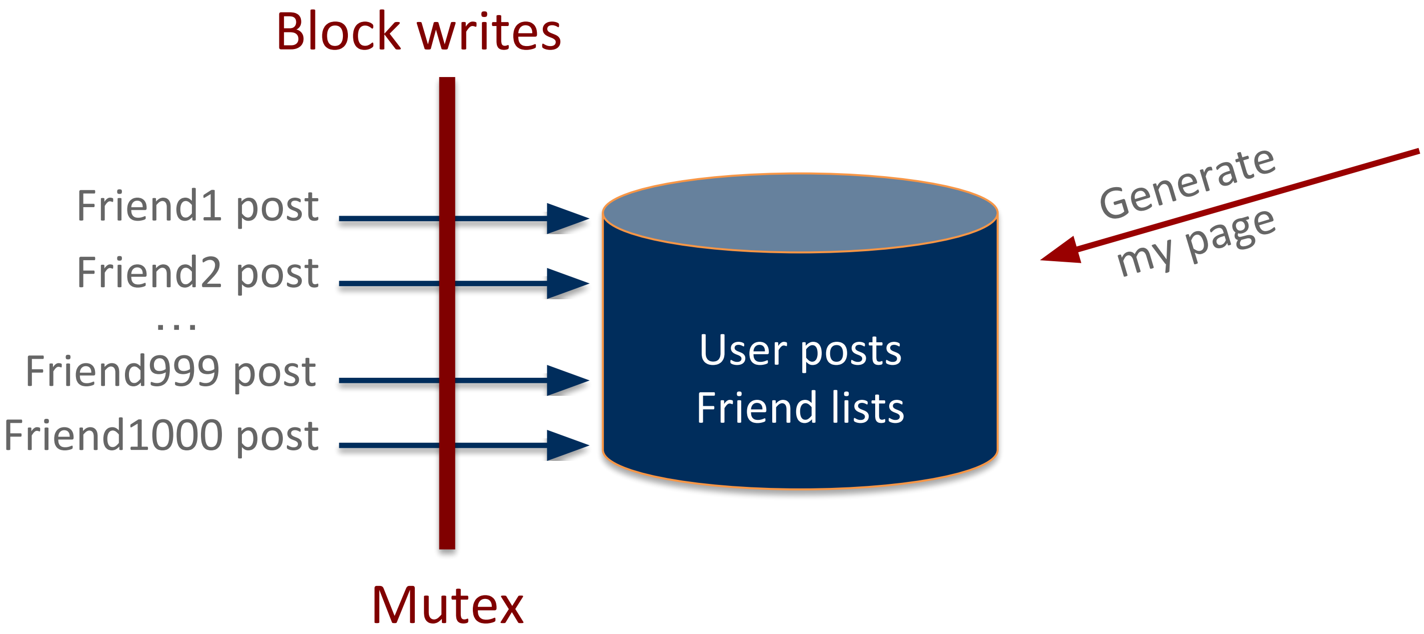
→ please be punctual 10.25am

Today's lecture: advanced concepts/ bringing it all together. Case studies are next.

Let’s Build a Distributed Database

E.g., as backend for a social network

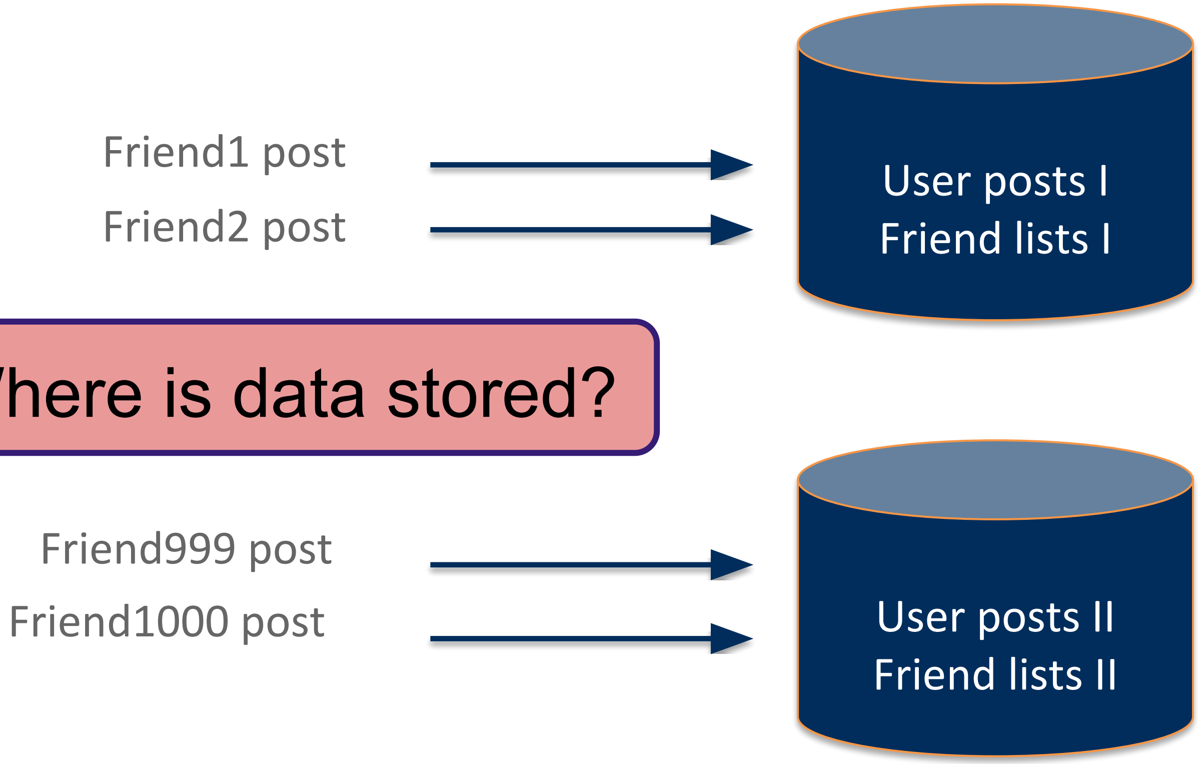
Single node:



Let’s Build a Distributed Database

E.g., as backend for a social network

Two nodes:

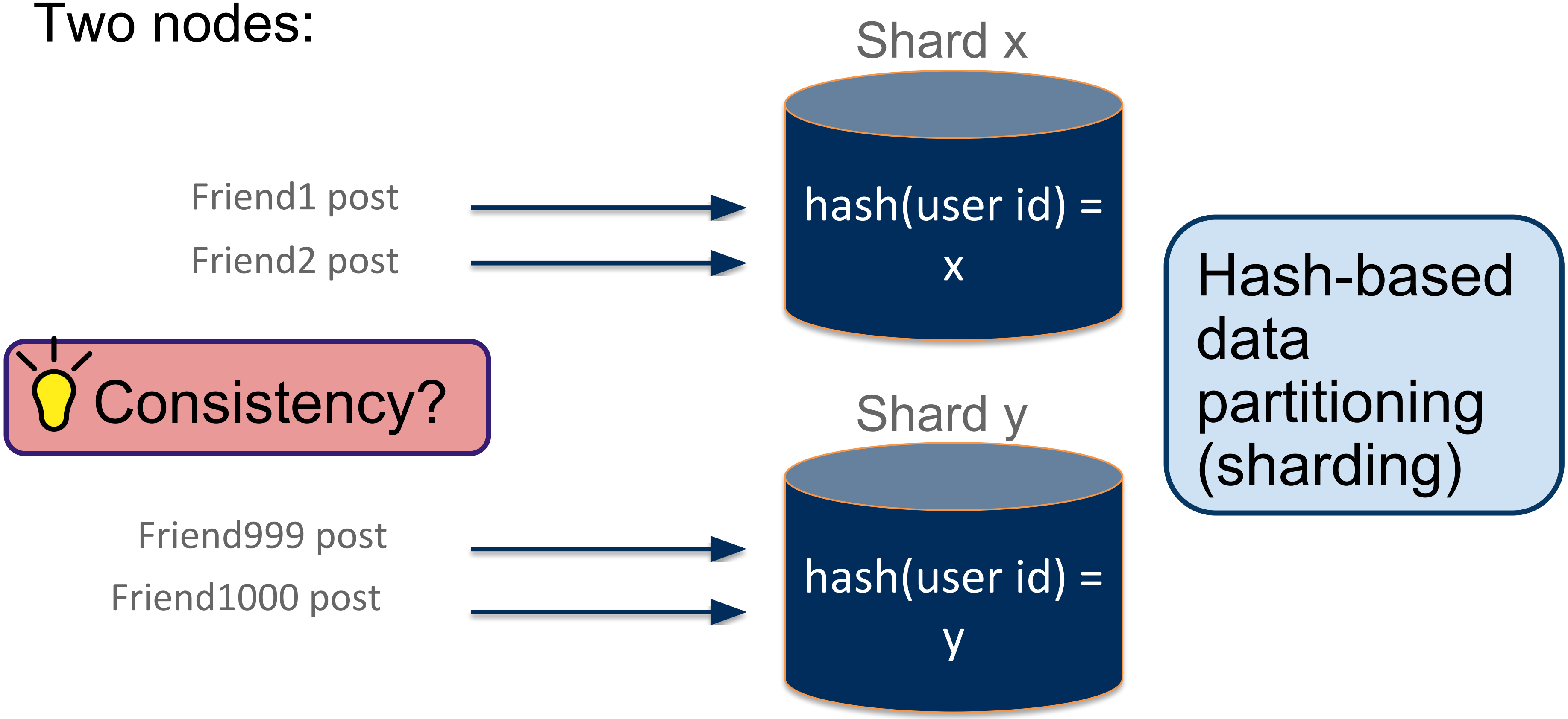


Where is data stored?

Let’s Build a Distributed Database

E.g., as backend for a social network

Two nodes:



Consistency?

Hash-based data partitioning (sharding)

Consistency Definitions

Sequential Consistency

- All nodes see operations in some sequential order
- Operations of each process appear in-order in this sequence

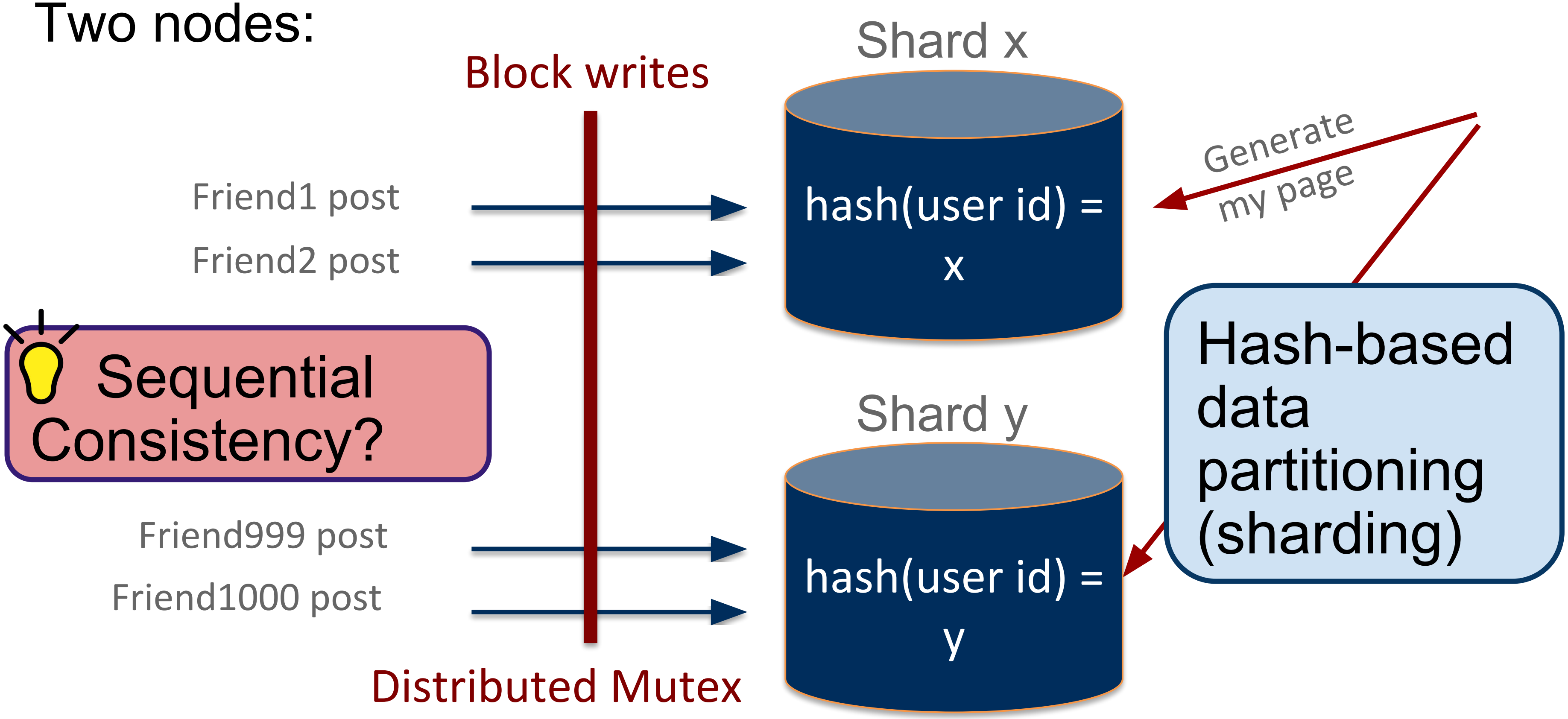
Eventual Consistency

- All nodes will learn eventually about all writes, in the absence of updates

Consistent Distributed Database

E.g., as backend for a social network

Two nodes:



Shard x

Block writes

Friend1 post

Friend2 post

hash(user id) = x

Generate my page

Sequential Consistency?

Shard y

Hash-based data partitioning (sharding)

Friend999 post

Friend1000 post

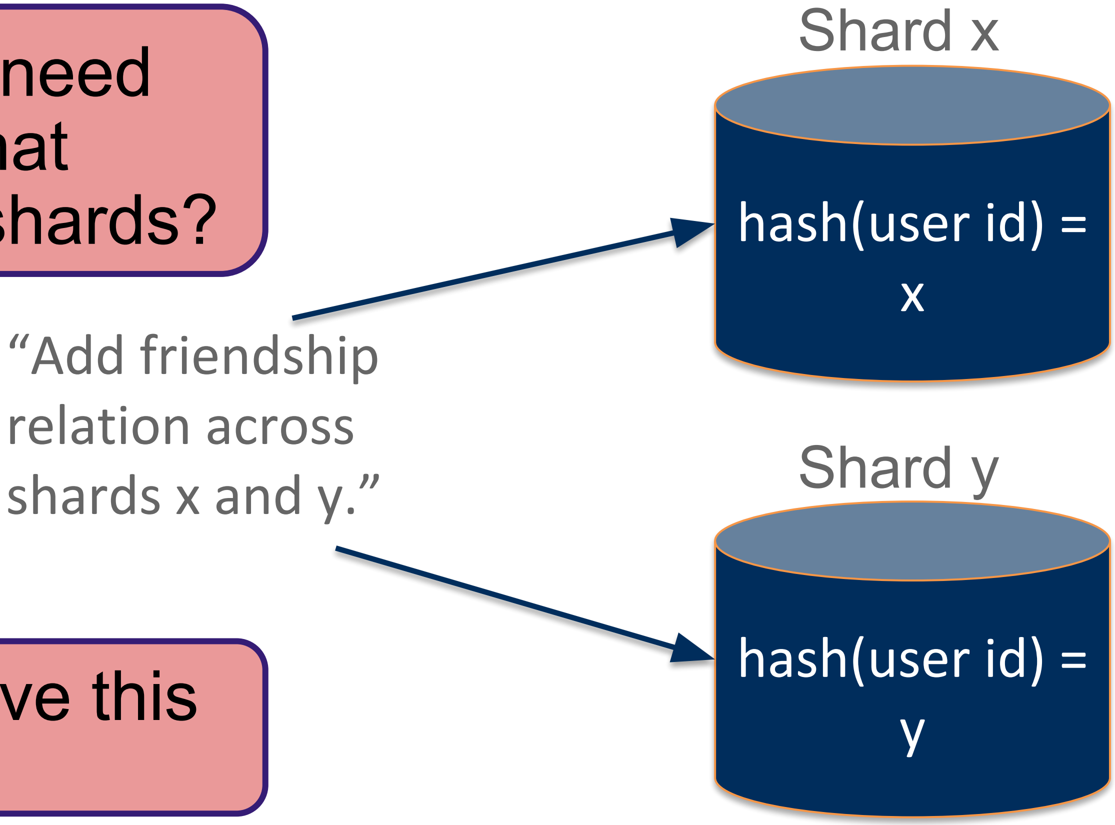
hash(user id) = y

Distributed Mutex

Distributed Database with Transactions

E.g., as backend for a social network

What if we need transactions that span several shards?



“Add friendship relation across shards x and y.”

Can you achieve this with a mutex?

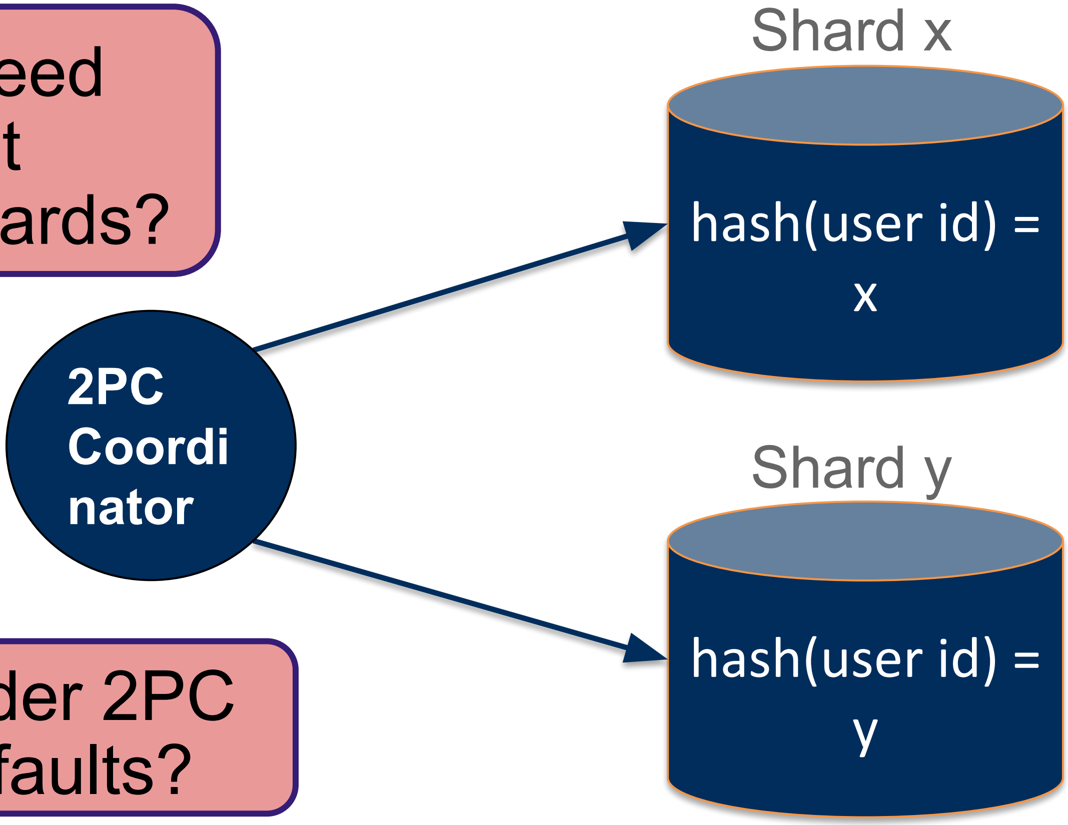
What would you use?

Distributed Database with Transactions

E.g., as backend for a social network

What if we need transactions that span several shards?

“Add friendship relation across shards x and y.”



Consistency under 2PC when there are faults?

What would you use to stay up during faults?

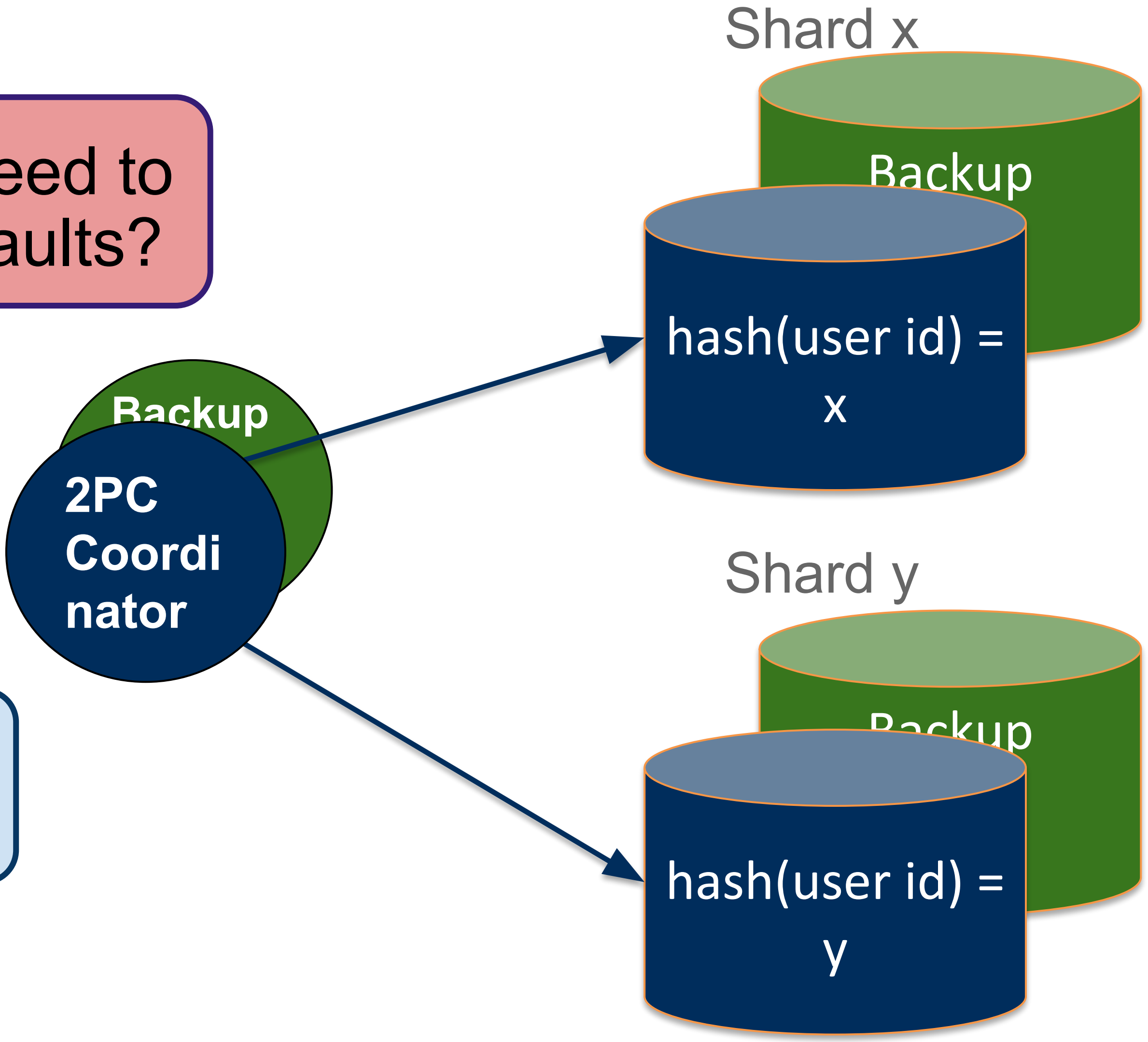
Fault-tolerant Distributed Database I

E.g., as backend for a social network

What if we need to stay up during faults?

"Add friendship relation across shards x and y."

Primary-Backup: Fail-over on fault



Backup

2PC Coordinator

Shard x

Backup

hash(user id) = x

Shard y

Backup

hash(user id) = y

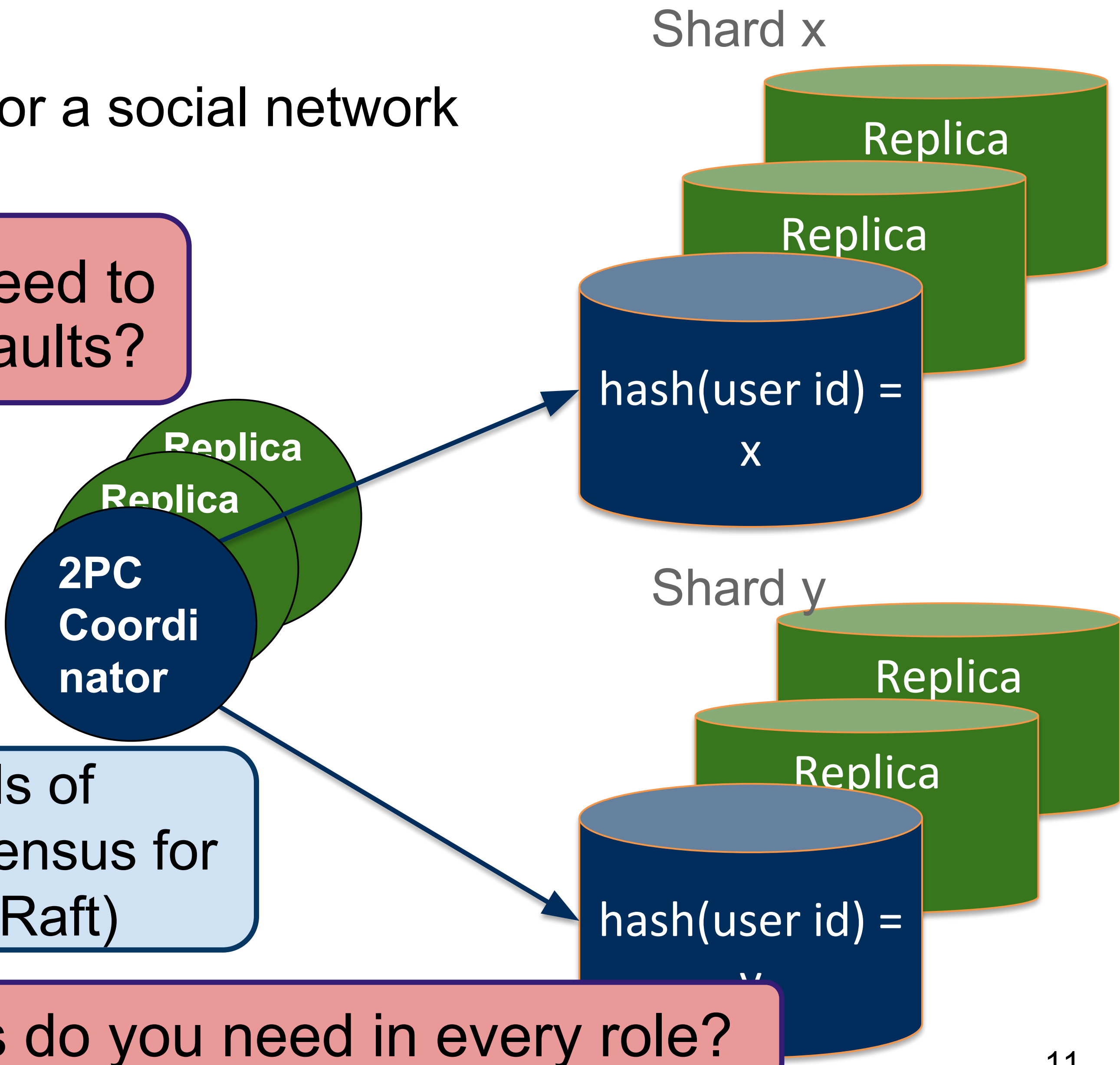
Fault-tolerant Distributed Database II

E.g., as backend for a social network

What if we need to stay up during faults?

“Add friendship relation across shards x and y.”

Run separate pools of replicas with consensus for every role (Paxos/Raft)



How many nodes do you need in every role?

Summary So Far: When to Use What?

	Use Case	**Problems**
Distributed Mutex	Distributed KV without transactions	
2PC	Distributed DB with transactions (e.g., Spanner)	
Primary-Backup	Cost-efficient fault tolerance (e.g., FaRM, GFS, VMWare-FT)	
Paxos	Staying up no matter the cost (e.g., Spanner, FaunaDB)	
RAID, Checksums	Every system	

Summary So Far: When to Use What?

	Use Case	**Problems**
Distributed Mutex	Distributed KV without transactions	Failures + Slow
2PC	Distributed DB with transactions (e.g., Spanner)	Failures
Primary-Backup	Cost-efficient fault tolerance (e.g., FaRM, GFS, VMWare-FT)	Correlated failures
Paxos	Staying up no matter the cost (e.g., Spanner, FaunaDB)	Delay and huge cost overhead
RAID, Checksums	Every system	Node failures

Practical Constraints

Can you think of cases where you would need a different solution than these algorithms?

- High performance: high throughput and low latency
  - Every consistency algorithm pays multiple RTTs!
- Availability during network partitions
  - Recall the CAP theorem
  - → When partitioned: either consistency (CP) or availability (AP)
- Simplicity and maintainability
  - 2018: still bugs in major consistency protocols [OSDI'18]

Different trade-offs made in practice
→ lectures with case studies today and after midterm

Practical Constraints: Alternative I

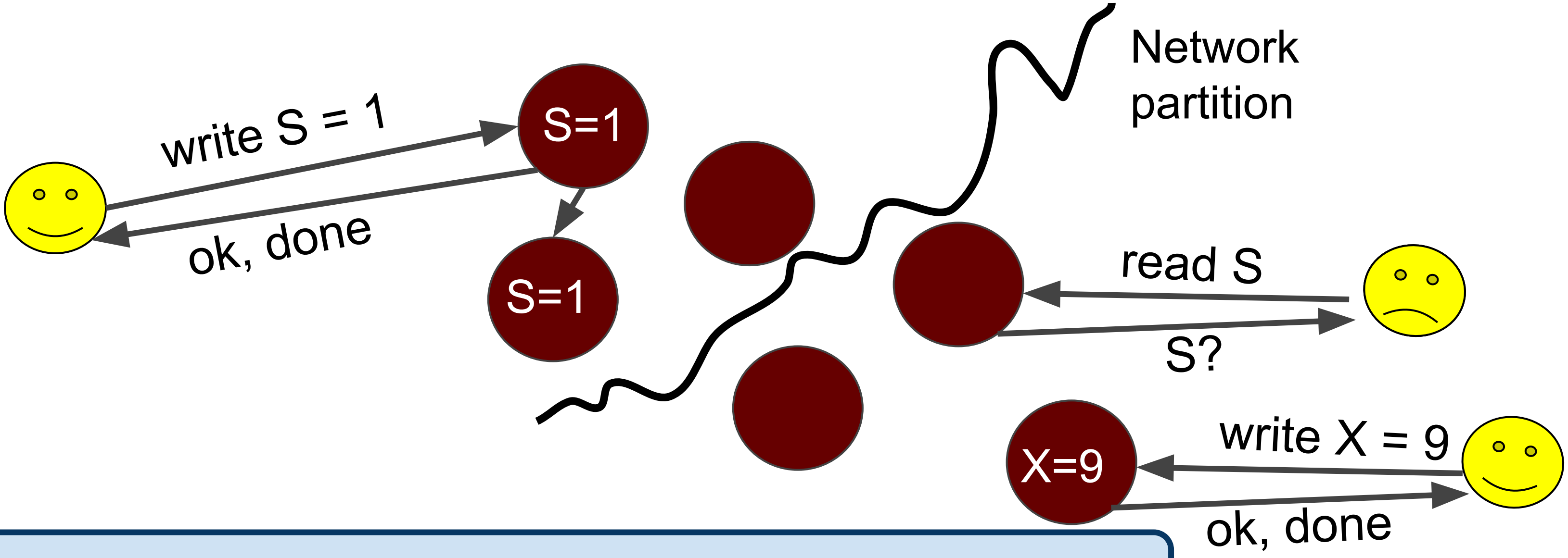
Only eventually consistent!

2005-2012: NoSQL systems

- Amazon’s Dynamo, Facebook’s Cassandra, Microsoft’s Azure CosmosDB, Apache CouchDB, Basho’s Riak

Design choices: AP: availability over consistency

“infinitely” scalable



Challenge: version reconcilation (parallel writes..)

Practical approach (Dynamo): Vector Clocks

Practical Constraints: Alternative II

2012-2018: resurgence of consistent distributed DBs

Google's Spanner, Microsoft's FaRM, Apple's FoundationDB
OSS: Calvin/FaunaDB, CockroachDB

Three key reasons [→ Daniel Abadi, UMD]

1. application code gets too complex and buggy without consistency support in DB
2. better network availability, CP (from CAP) choice is more practical, availability sacrifice hardly noticeable
3. CAP asymmetry: CP can guarantee consistency, AP can't guarantee availability (only question of degree)

Trend: stronger-than-sequential consistency

Consistency Definitions

External Consistency

- If T1 commits before T2, then the commit order must be T1 before T2

Sequential Consistency

- All nodes see operations in some sequential order
- Operations of each process appear in-order in this sequence

Eventual Consistency

- All nodes will learn eventually about all writes, in the absence of updates

Consistency matters for a DB

- Example in social network database

Two transactions:

1. Remove untrustworthy person X as friend
2. Post P: “My government is repressive…”

💡 What if commit order T2 before T1?

💡 We often need external consistency!

Practical Constraints: Alternative II

2012-2018: resurgence of consistent distributed DBs

Google's Spanner, Microsoft's FaRM, Calvin and FaunaDB

These guarantee at least sequential consistency, unlike NoSQL.

Three key reasons [→ Daniel Abadi, UMD]

1. application code gets too complex and buggy without consistency support in DB
2. better network availability, CP (from CAP) choice is less relevant, availability sacrifice hardly noticeable
3. CAP asymmetry: CP can guarantee consistency, AP can't guarantee availability (only question of degree)

Even stronger consistency requirements.

Most workloads are read heavy. New systems support lock-free consistent reads.

Revisiting the CAP Theorem

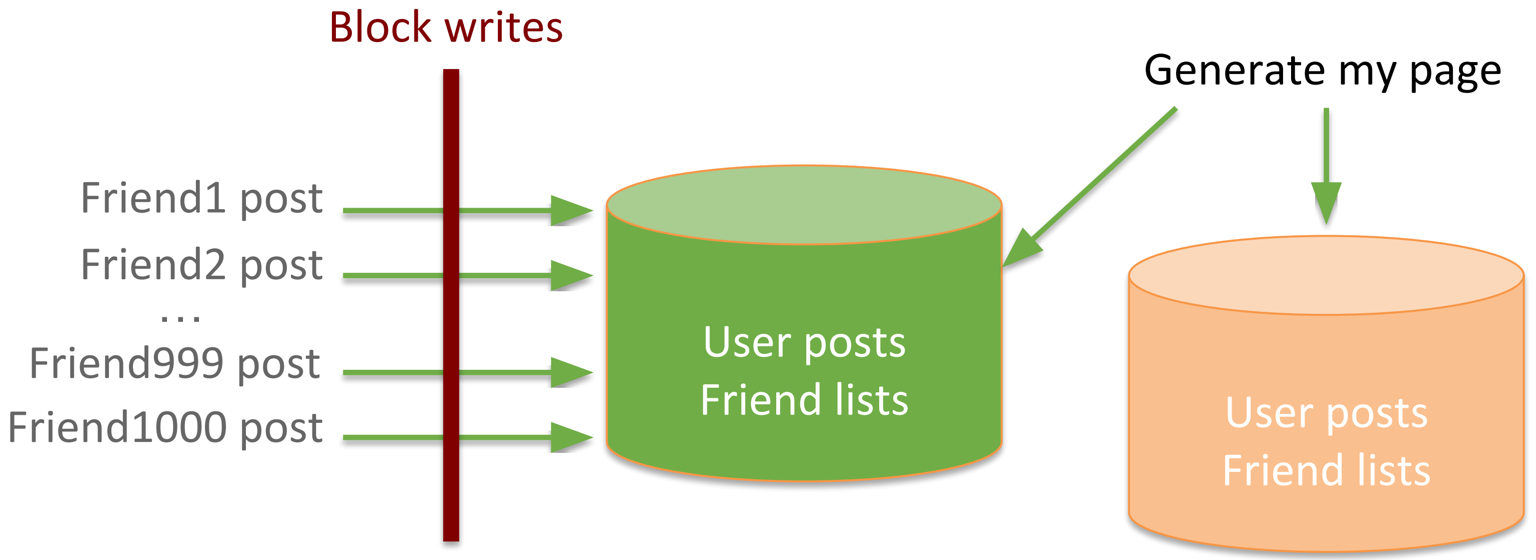
Daniel Abadi, UMD]

“The CAP theorem says that it is impossible for a system that guarantees consistency to guarantee 100% availability in the presence of a network partition.”

No system can guarantee 100% availability in practice! So, can’t guarantee A.

Rather, guaranteeing consistency causes a reduction to our already imperfect availability.

Reading from Single Machine



Block writes

Generate my page

Friend1 post

Friend2 post

...

Friend999 post

Friend1000 post

User posts
Friend lists

User posts
Friend lists

Read lock

Block all writes until read has finished

Snapshot

Read from DB-copy,

writes continue to original DB

Lock free ideas?

Figures adapted from [Wilson Hsieh and coauthors, OSDI 2012]

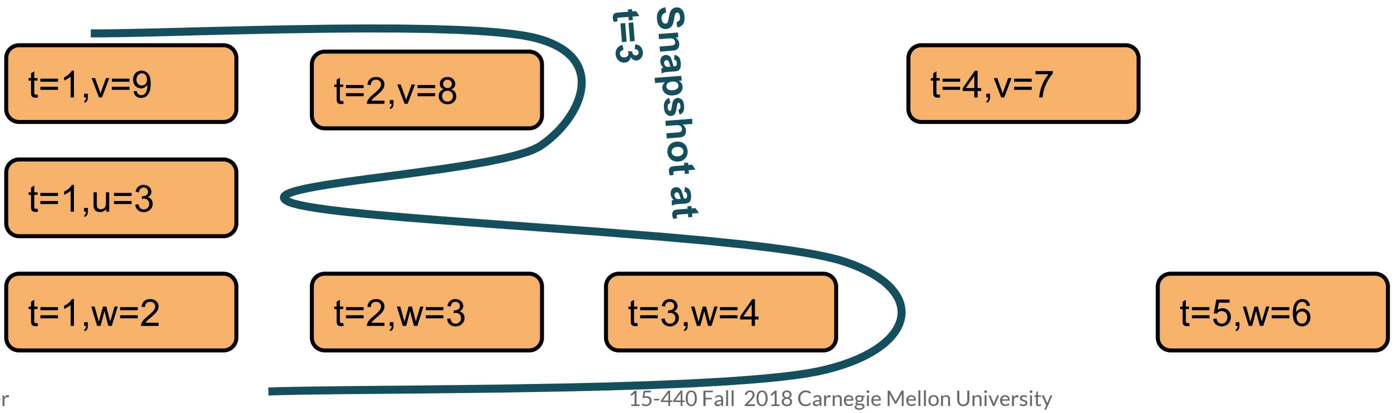
Implementing Snapshot Reads

Actually make a **“copy”**

- How do you deal with concurrency?

Multi-version concurrency control

- New commit → add as (timestamp,value)
- Keep old (timestamp,value) tuples
- Snapshot: read latest tuples with timestamp < now



t=1,v=9

t=2,v=8

t=4,v=7

t=1,u=3

t=1,w=2

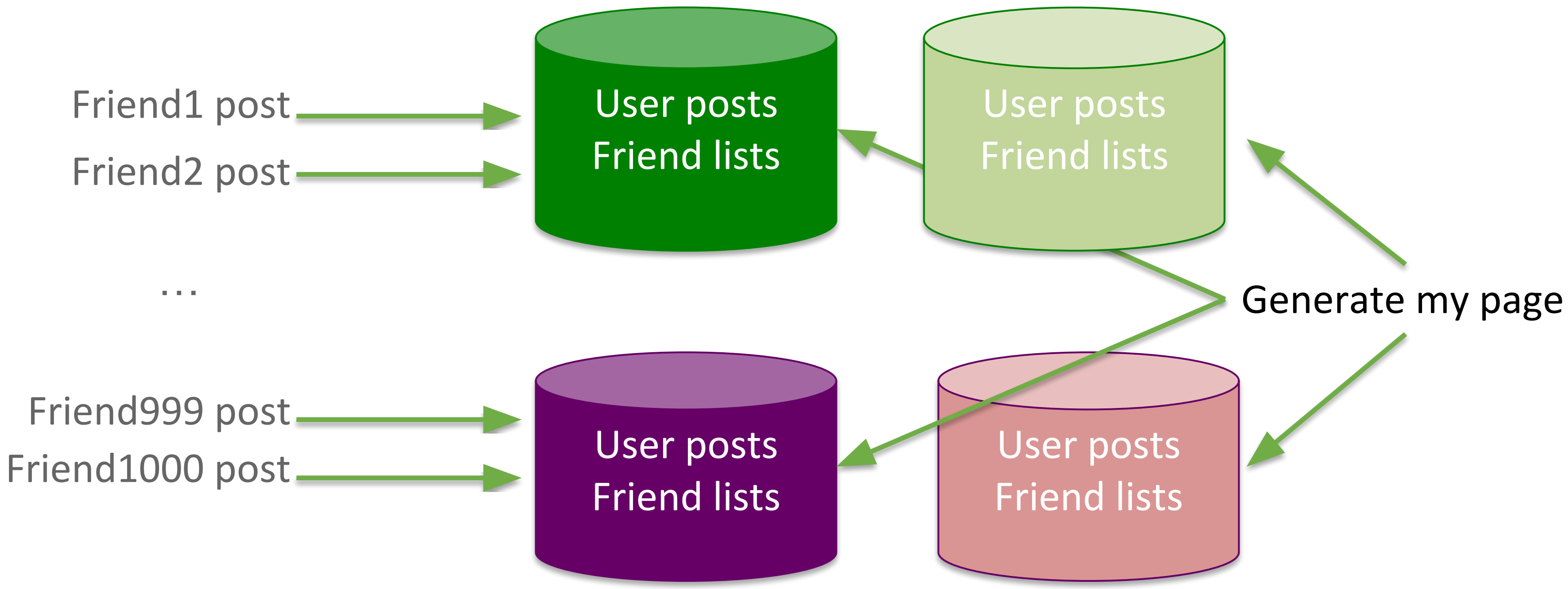
t=2,w=3

t=3,w=4

t=5,w=6

Snapshot at t=3

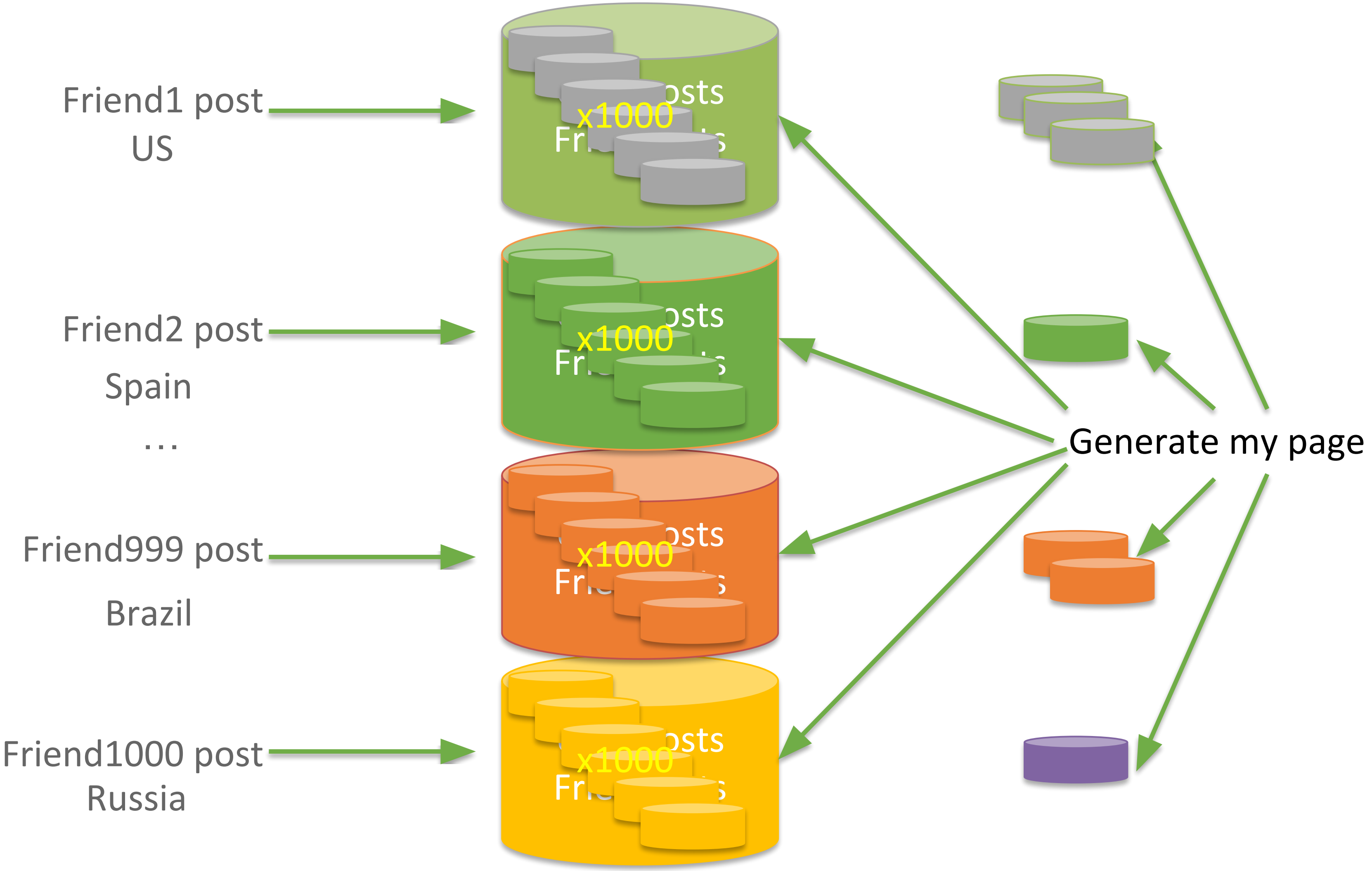
Reading from Multiple Machines



Snapshot

Requirement: create distributed snapshots at exactly the same time!

Real-World Distributed DB



Friend1 post
US

Friend2 post
Spain

…

Friend999 post
Brazil

Friend1000 post
Russia

Generate my page

Multi-Version Databases

Widely implemented

In advanced single-node RDBMs

PostgreSQL

💡 Challenge in distributed DBs?

Need **synchronized clocks** across all nodes

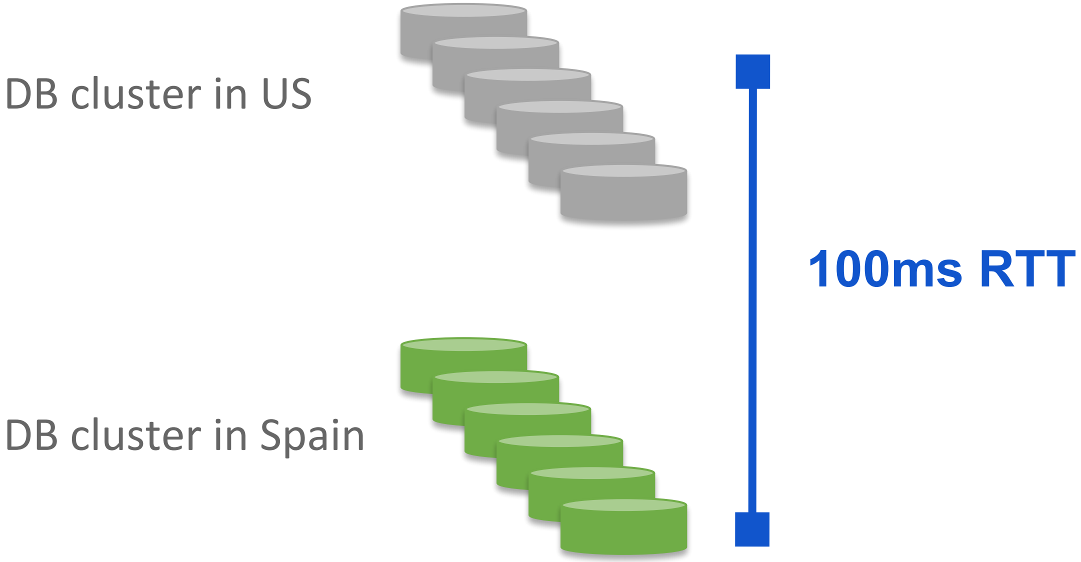
So, what!? We learned how to sync time in 440!

Need highly accurate time synchronization

e.g., 1,000,000 reqs/sec ⟶ error < 1 microsecond

What do we know about time sync errors?

Time Synchronization Error



Time sync error proportional to RTT

Global Internet RTTs in 100s of milliseconds

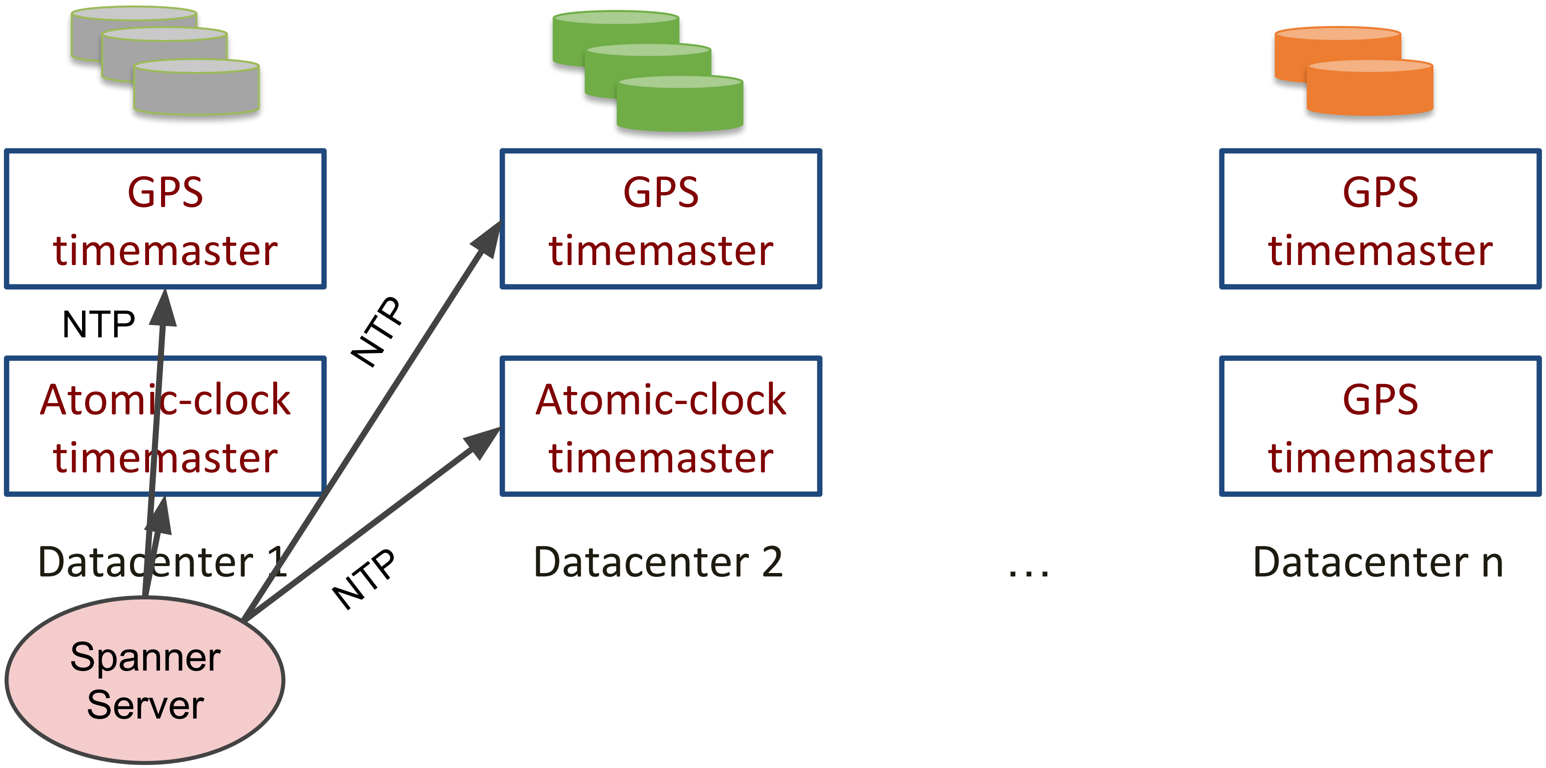
⇒ No microsecond time sync protocol across Internet

Spanner: Google's Globally-Distributed Database

- Feature: **Lock-free** distributed **read** transactions
- Property: **External consistency** of distributed transactions
  - First externally consistent DB at global scale
- Implementation: WAL + 2PC + Paxos + Snapshots
- Enabling technology: **TrueTime**
  - Interval-based global time

OSDI 2012

How does Spanner do Time Sync?



GPS timemaster

Atomic-clock timemaster

Datacenter 1

GPS timemaster

Atomic-clock timemaster

Datacenter 2

…

GPS timemaster

GPS timemaster

Datacenter n

Spanner Server

NTP

NTP

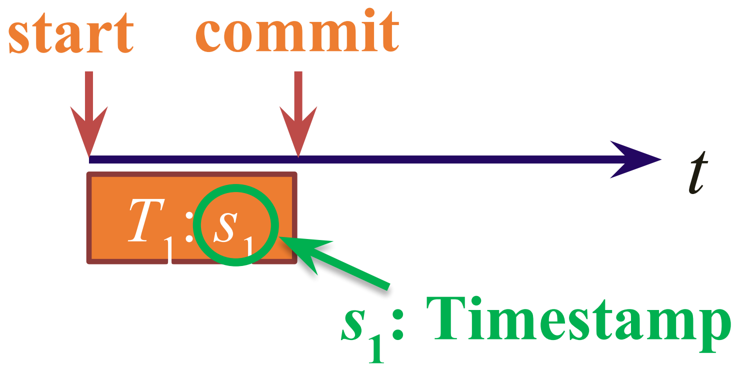
NTP

...The majority of masters have GPS receivers with dedicated antennas... The remaining masters (which we refer to as Armageddon masters) are equipped with atomic clocks. An atomic clock is not that expensive: the cost of an Armageddon master is of the same order as that of a GPS master...

OSDI 2012

Timestamps and Concurrency Control

- Key aspect: globally meaningful timestamps for distributed transactions



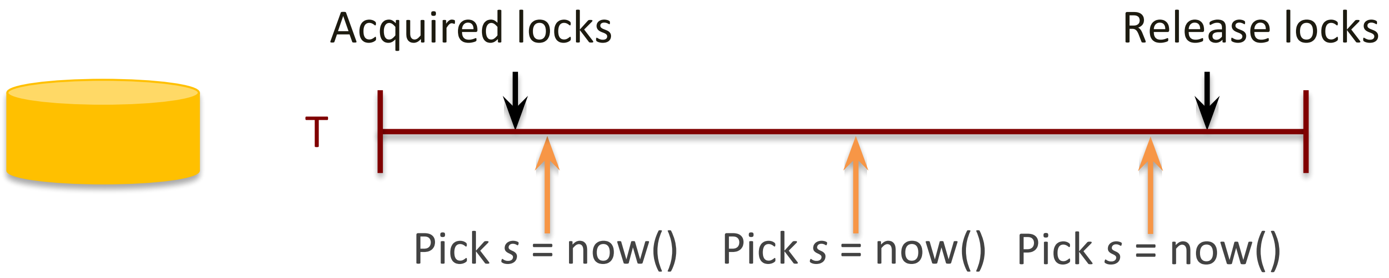
start commit

t

$T_1 : s_1$

s_1: Timestamp

- Strict two-phase locking for write transactions
- Assign timestamp while locks are held



Acquired locks

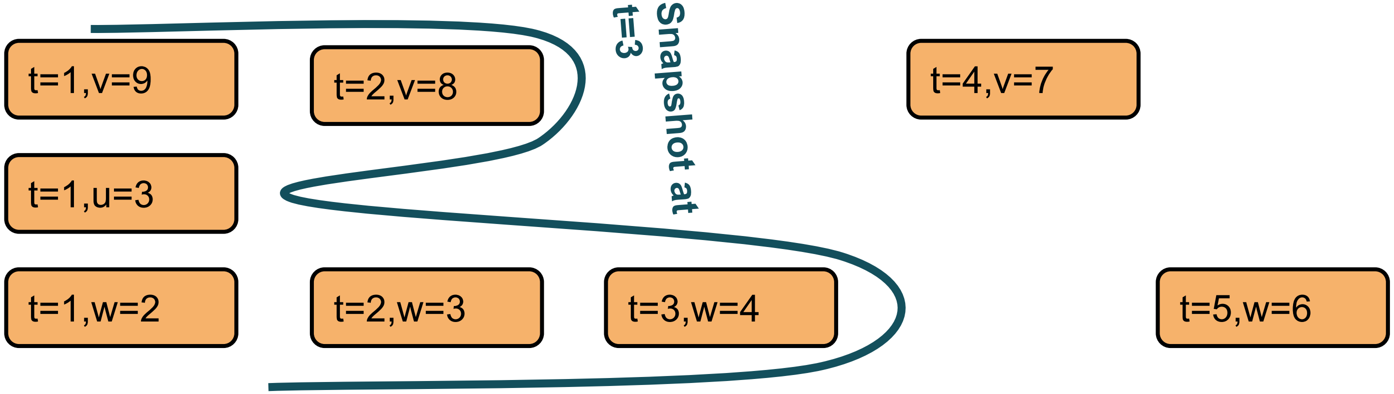
Release locks

T

Pick s = now() Pick s = now() Pick s = now()

OSDI 2012

Sufficient to Assign Timestamps?



t=1,v=9

t=2,v=8

Snapshot at t=3

t=4,v=7

t=1,u=3

t=1,w=2

t=2,w=3

t=3,w=4

t=5,w=6

Challenge: time sync errors even with GPS/atomic clocks

Conceptually: must wait until all write transactions visible (their timestamps have passed)

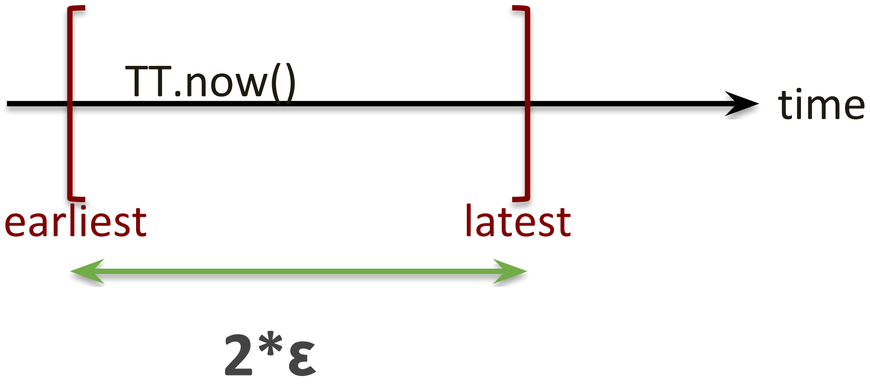
Key question: how long do we need to wait?

What is the clock uncertainty (worst time sync error?)

OSDI 2012

Spanner’s TrueTime Concept

- “Global wall-clock time” with bounded uncertainty



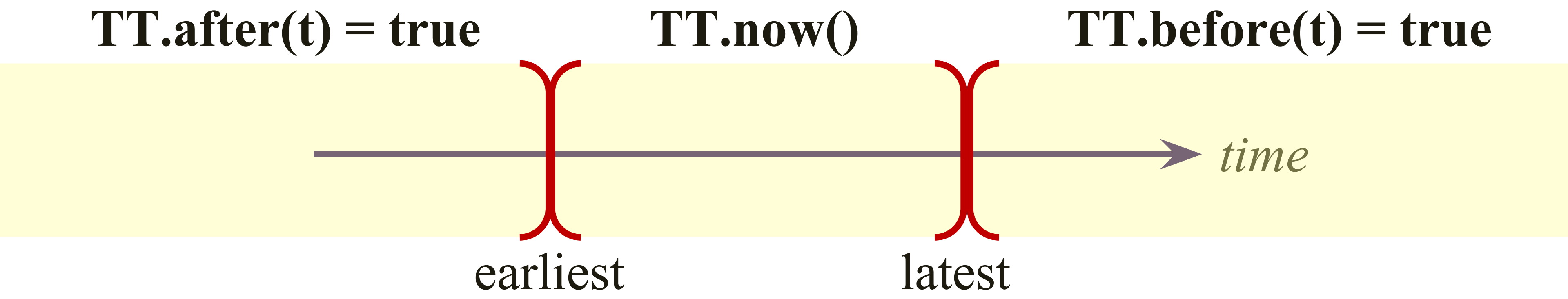
TT.now()

time

earliest

latest

2*ε



TT.after(t) = true

TT.now()

TT.before(t) = true

time

earliest

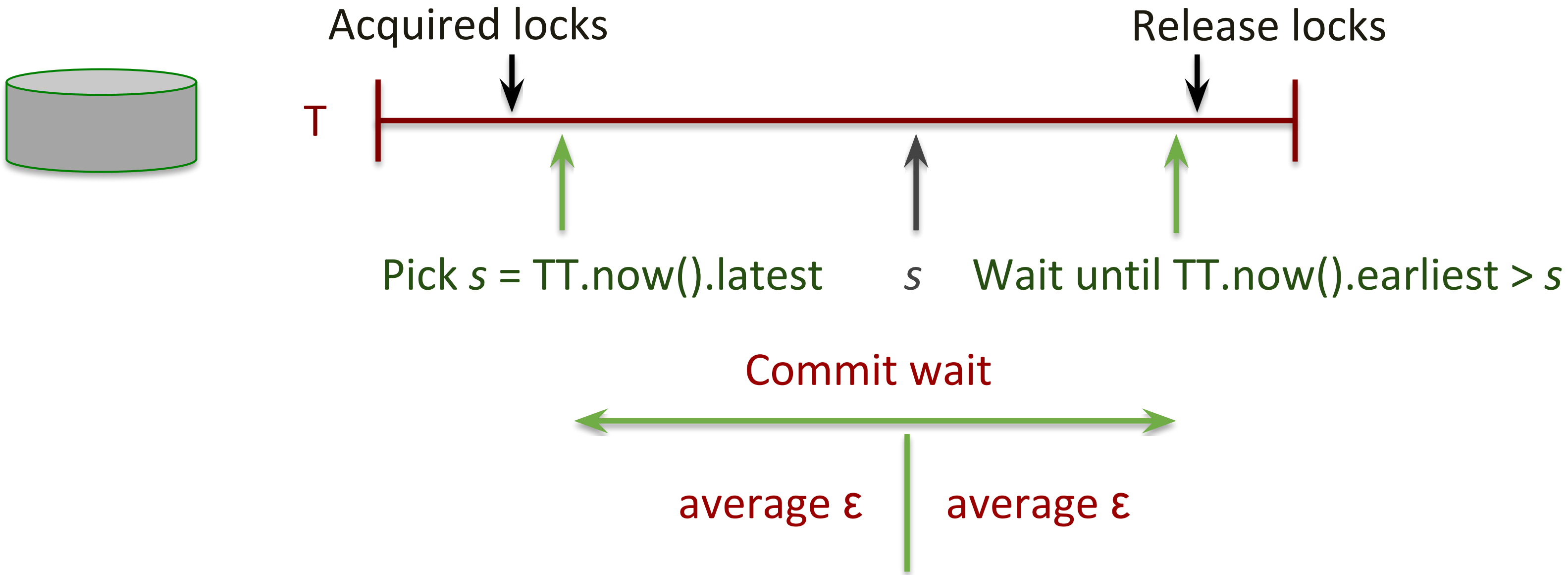
latest

TT.after(t) – true if **t** has definitely passed

TT.before(t) – true if **t** has definitely not arrived

OSDI 2012

Timestamps and TrueTime



Acquired locks

Release locks

T

Pick *s* = TT.now().latest

s

Wait until TT.now().earliest > *s*

Commit wait

average ε

average ε

Why set s=latest and why wait until earliest>s?

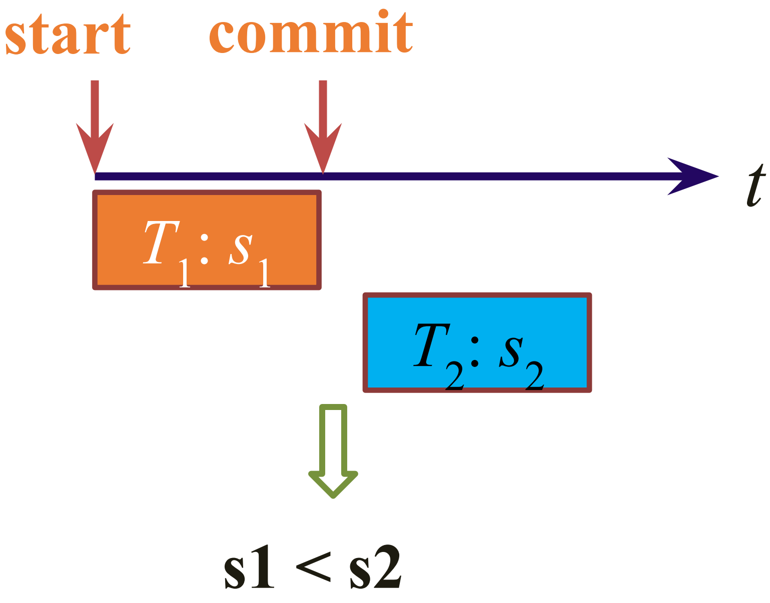
T1 starts at t=2 ε=2
picks s=4
releases locks at t=6

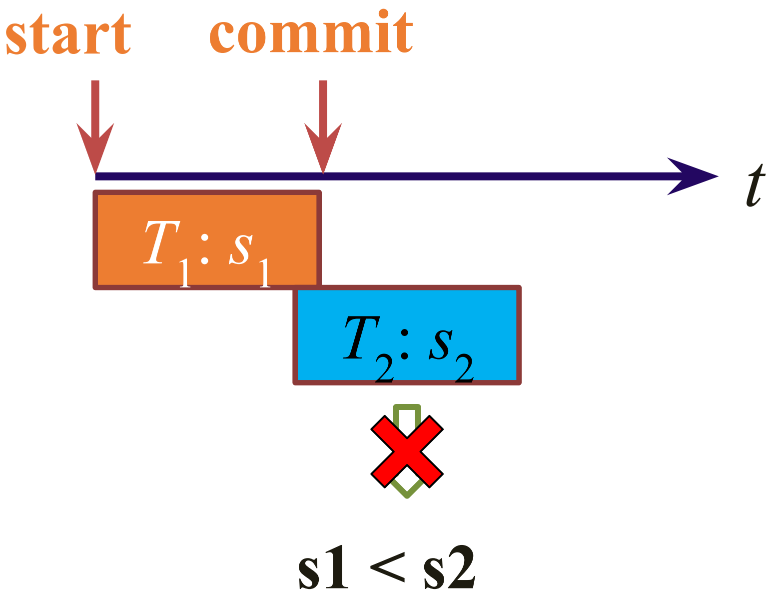
T2 starts at t=6 ε=2
picks s=8
releases locks at t=10

T3 starts at t=8 ε=3
picks s=11
releases locks at t=14

Spanner External Consistency

- If a transaction T_1 commits before another transaction T_2 starts, then T_1's commit timestamp is smaller than T_2
- Similar to how we reason with wall-clock time





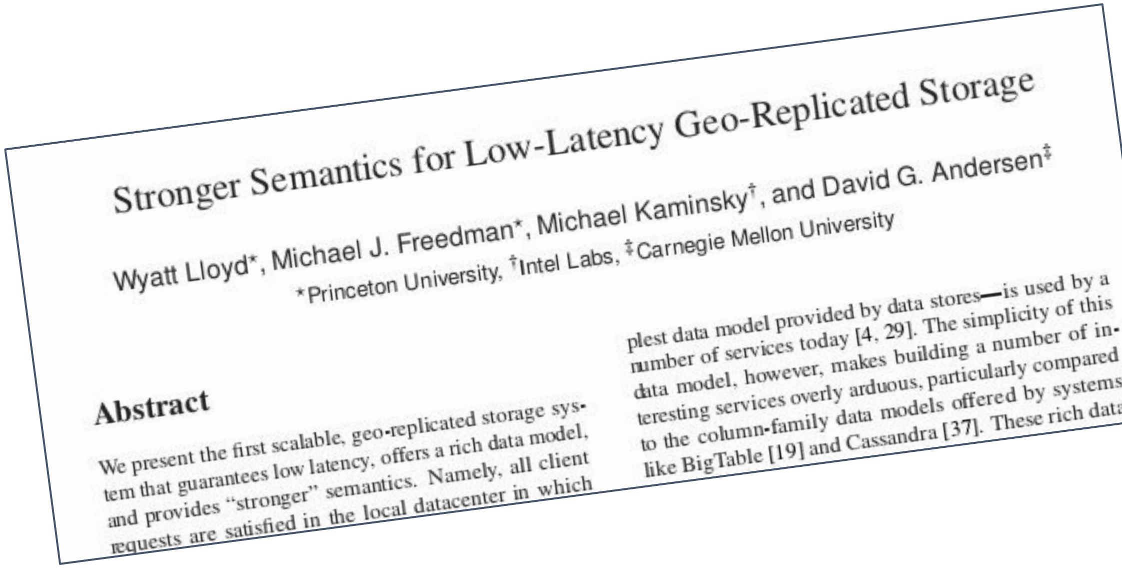
Spanner Summary

- Globally consistent replicated database system
- Implements distributed transactions
  - Uses 2PC
- Fault-tolerance and replicated writes
  - Uses Paxos based
- Newer Systems:

Cloud Spanner

FoundationDB

Cockroach DB



Stronger Semantics for Low-Latency Geo-Replicated Storage

Wyatt Lloyd[*], Michael J. Freedman[*], Michael Kaminsky[†], and David G. Andersen[‡]

[*]Princeton University, [†]Intel Labs, [‡]Carnegie Mellon University

Abstract

We present the first scalable, geo-replicated storage system that guarantees low latency, offers a rich data model, and provides "stronger" semantics. Namely, all client requests are satisfied in the local datacenter in which

plest data model provided by data stores—is used by a number of services today [4, 29]. The simplicity of this data model, however, makes building a number of interesting services overly arduous, particularly compared to the column-family data models offered by systems like BigTable [19] and Cassandra [37]. These rich data