Building Geo-Distributed Large-scale Systems

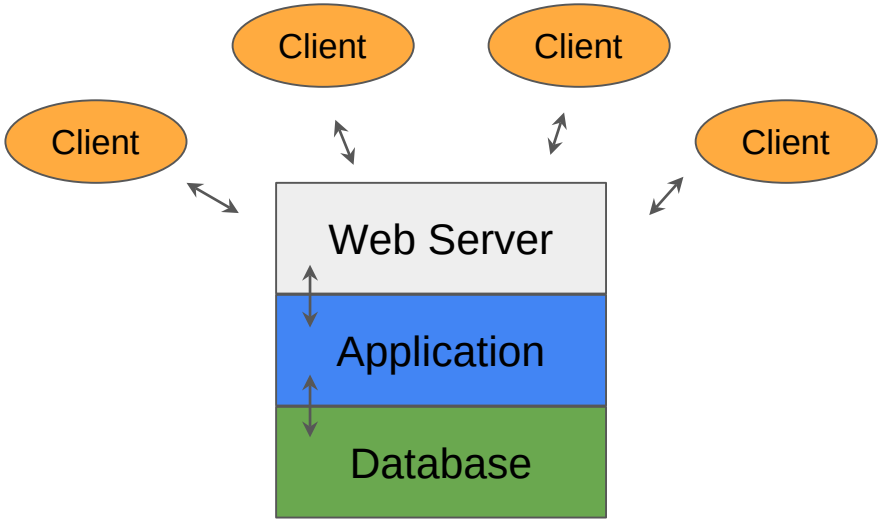
Guest Lecture

Outline

- Architecture of a Distributed Application
- Memcached at Facebook [NSDI'13]
- Spanner at Google [OSDI'12]
- Takeaways

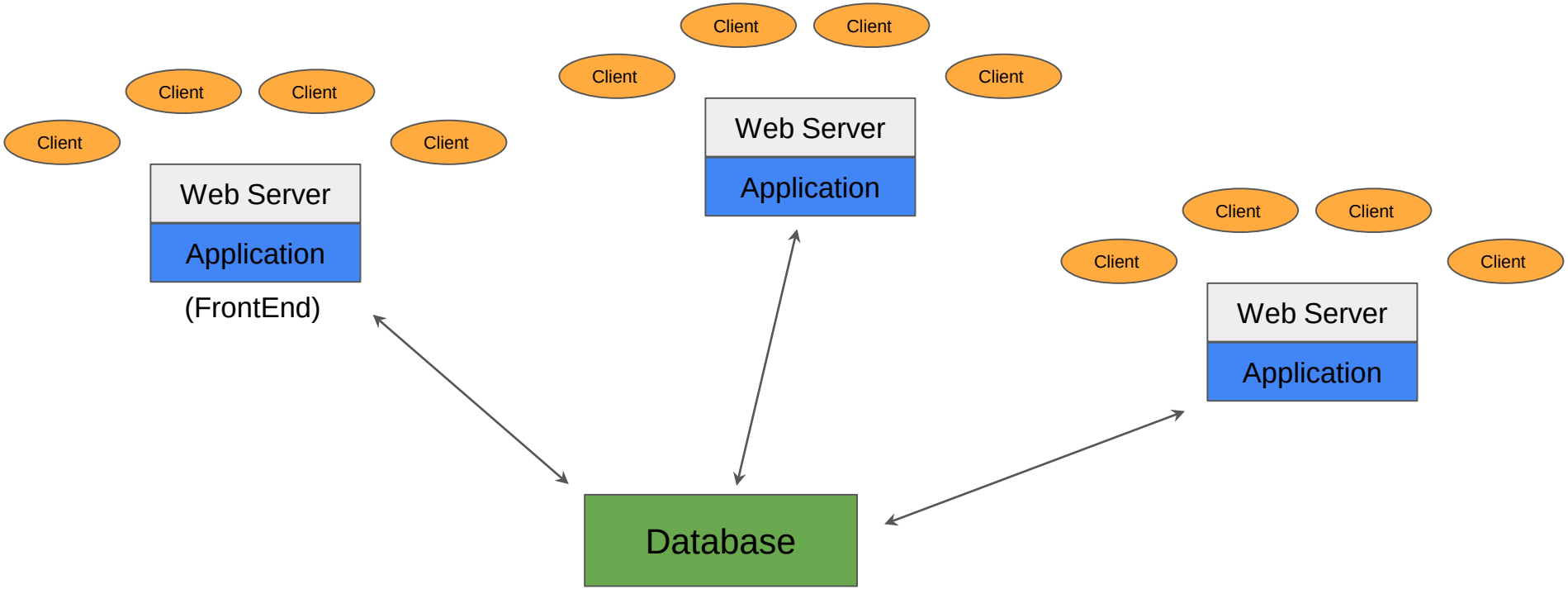
A basic web application

- Database: Stores all the data in a persistent manner.
- Application: Contains the logic of generating client responses from the data.
- Web Server: Interacts with clients, responding to their requests.

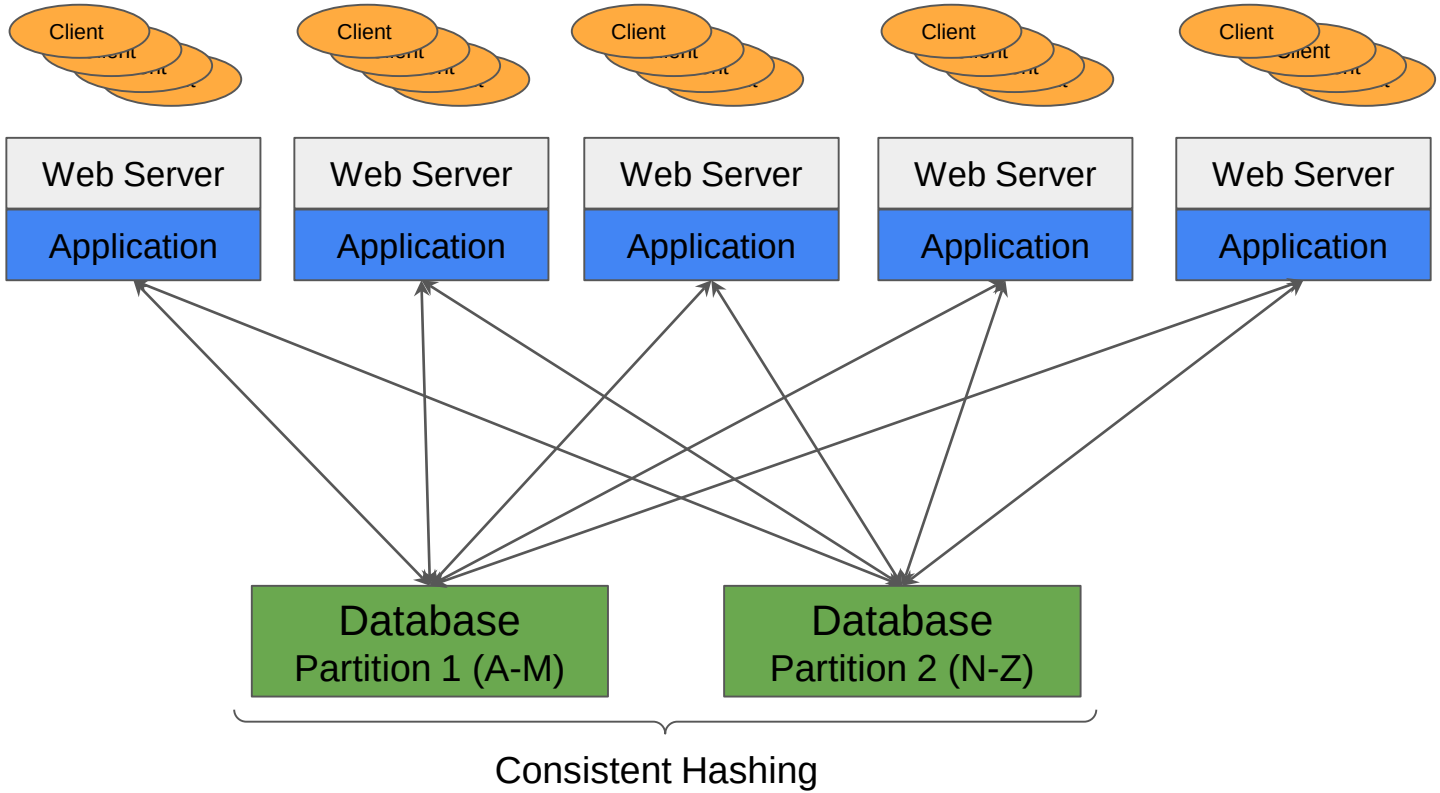


Scaling up

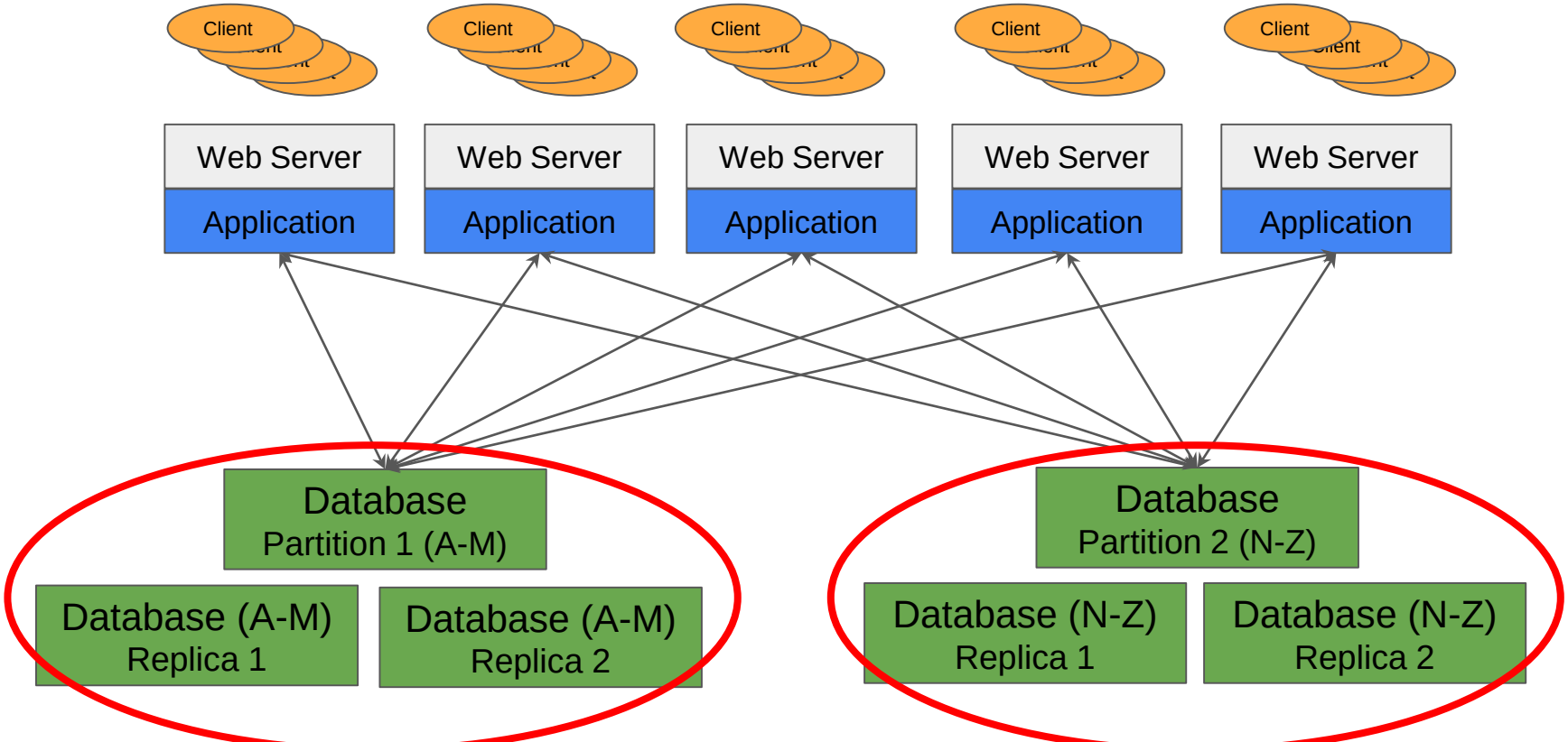
Single machine insufficient to handle CPU load.



Scaling up even more



Handling Failures



Replication

Could be of different types (depending of what failures we want to handle)

- Within the same cluster
  - Provides tolerance against single machine failures
- Within the same datacenter (but different clusters)
  - Provides tolerance against cluster failures -- e.g., network switch failures
- Within the same zone (but different datacenters)
  - Provides tolerance against datacenter failures
- Across zones, or geo-distributed
  - Provides tolerance against catastrophic failures -- e.g. earthquakes, hurricanes

Another huge benefit of replication

Low latency!

Client requests can be routed to the nearest data center.

But how? Shouldn't www.google.com point to a single IP address?

Typically done using GeoDNS (Domain Name System) -- where a name resolves to closest replica.

For e.g., I get the following IP address when resolving www.google.com

```
$ nslookup www.google.com

www.google.com.   4        IN       A        172.217.5.100
```

Outline

- Architecture of a Distributed Application
- Memcached at Facebook
- Spanner at Google
- Takeaways

Case study of two geo-distributed systems

Facebook's Memcached

A distributed key-value store that supports the FB's social network. The system handles billions of requests per second and holds trillions of items.

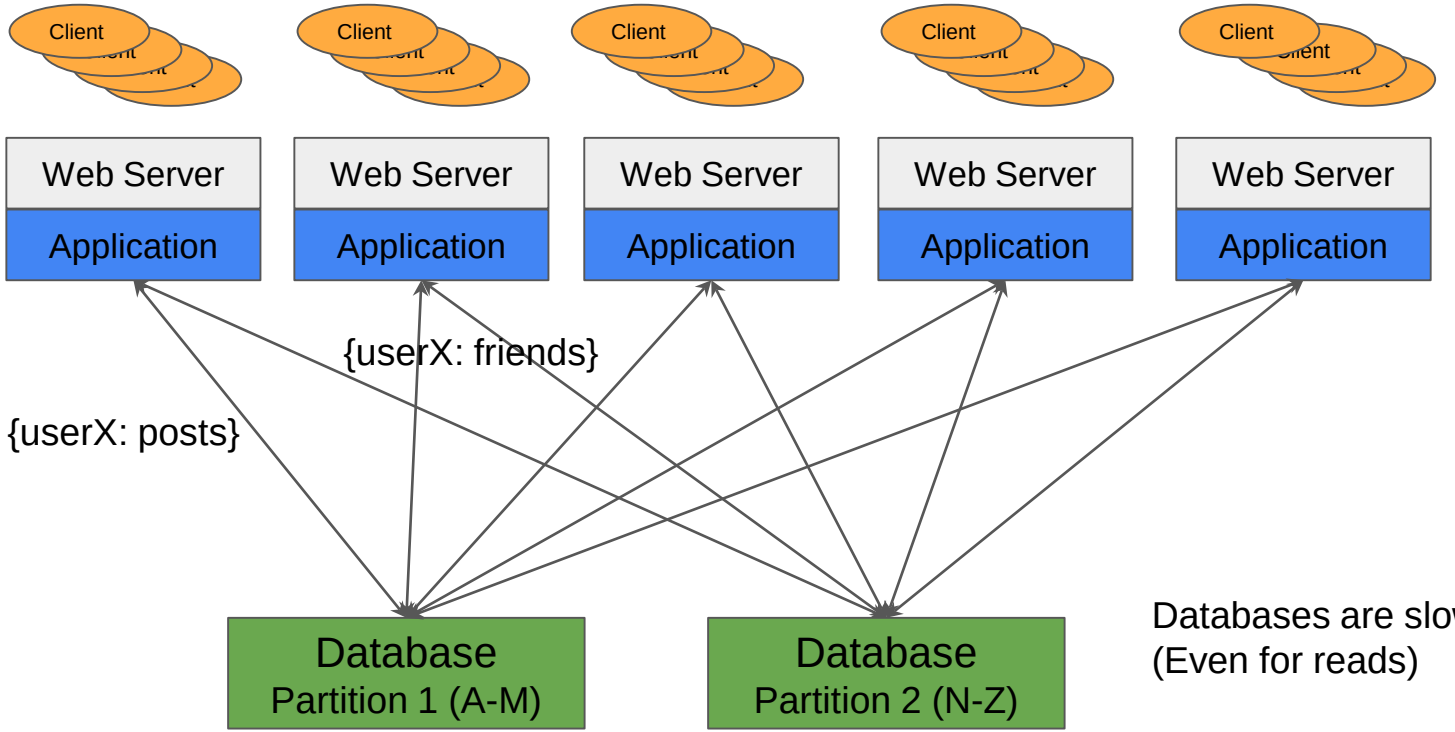
- How they build super high capacity from mostly-off-the-shelf software.
- Address the fundamental struggle between performance and consistency

Google's Spanner

Google’s scalable, multi-version, globally distributed, and synchronously replicated database.

- How they provide strong consistency without sacrificing performance*.
- A rare example of wide-area synchronous replication.

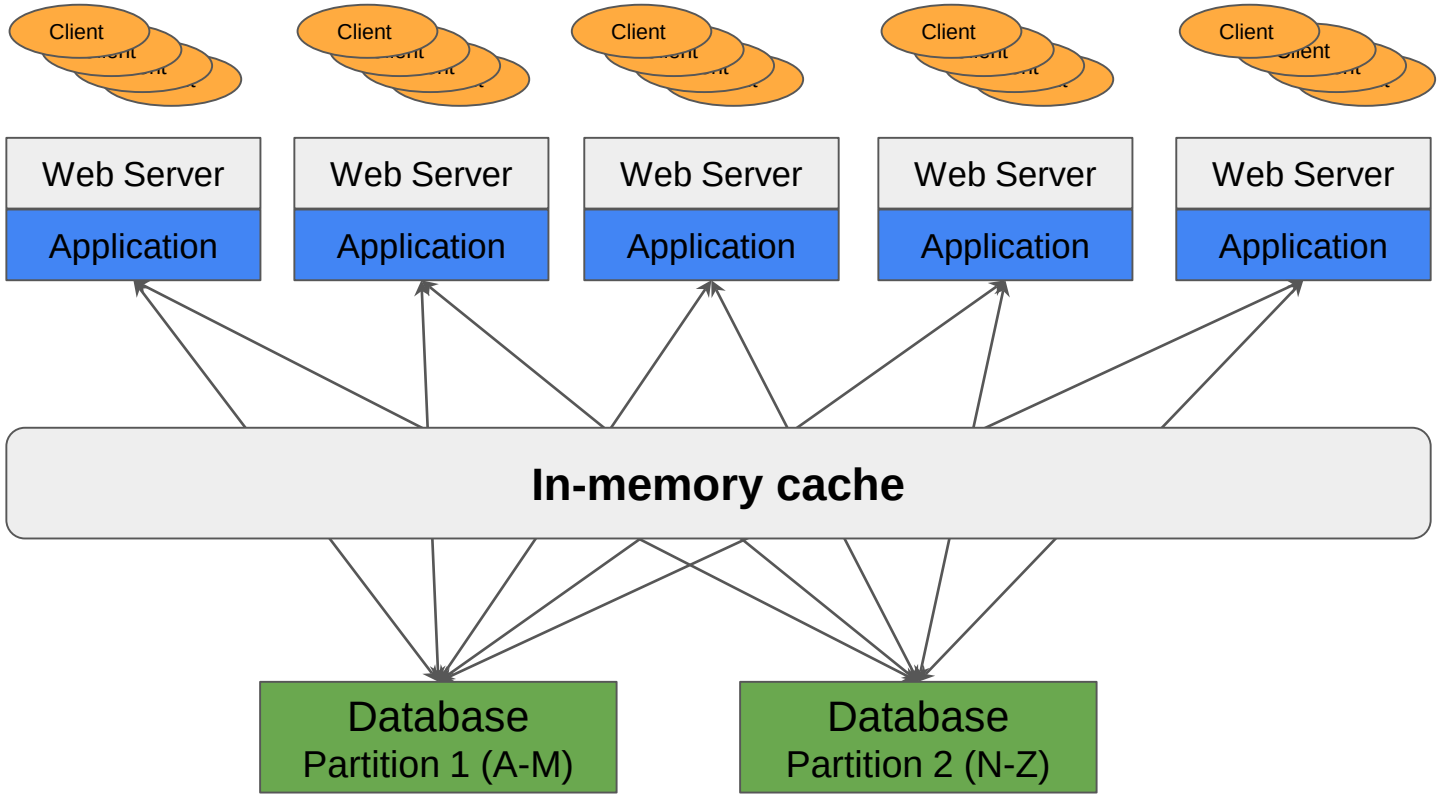
Facebook's Memcached Architecture



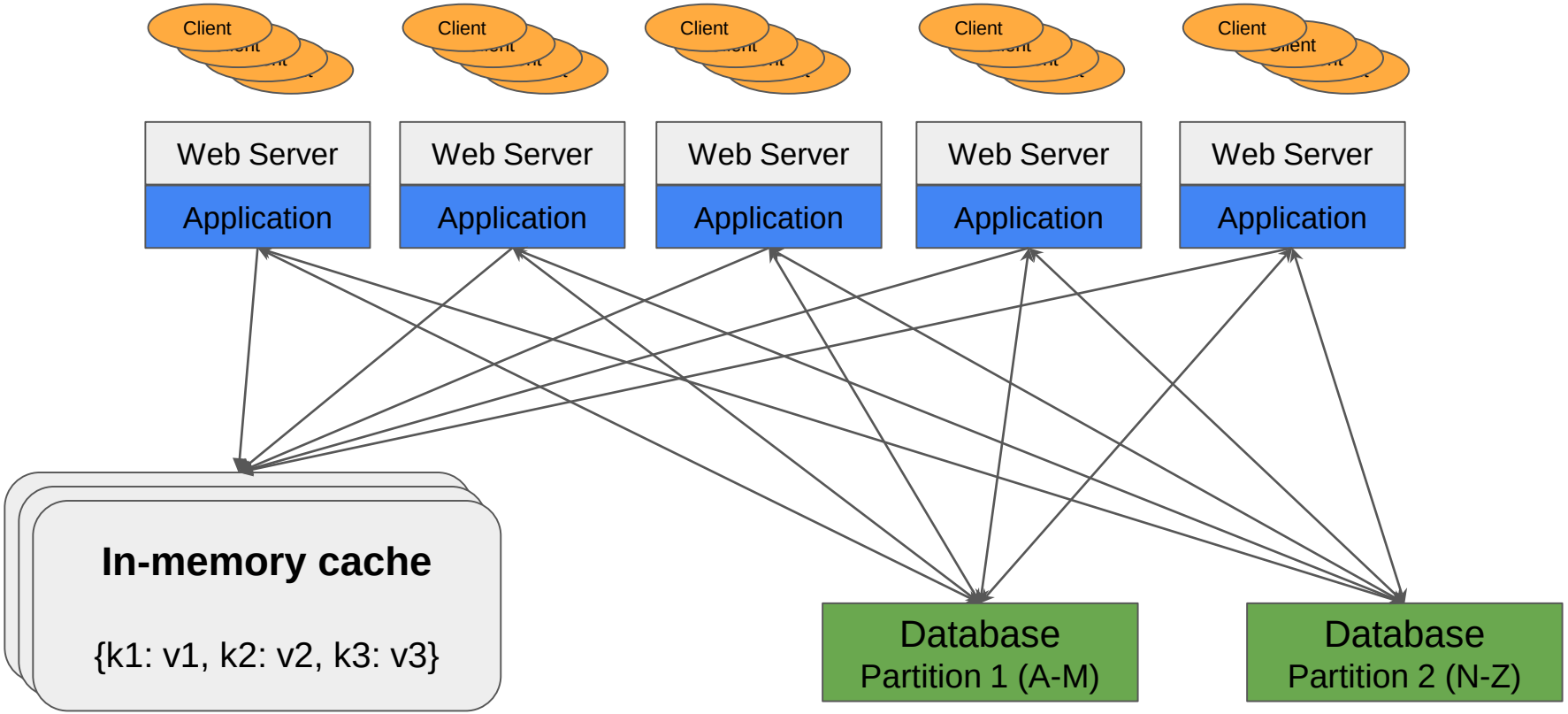
Databases are slow!
(Even for reads)

One key is very popular?

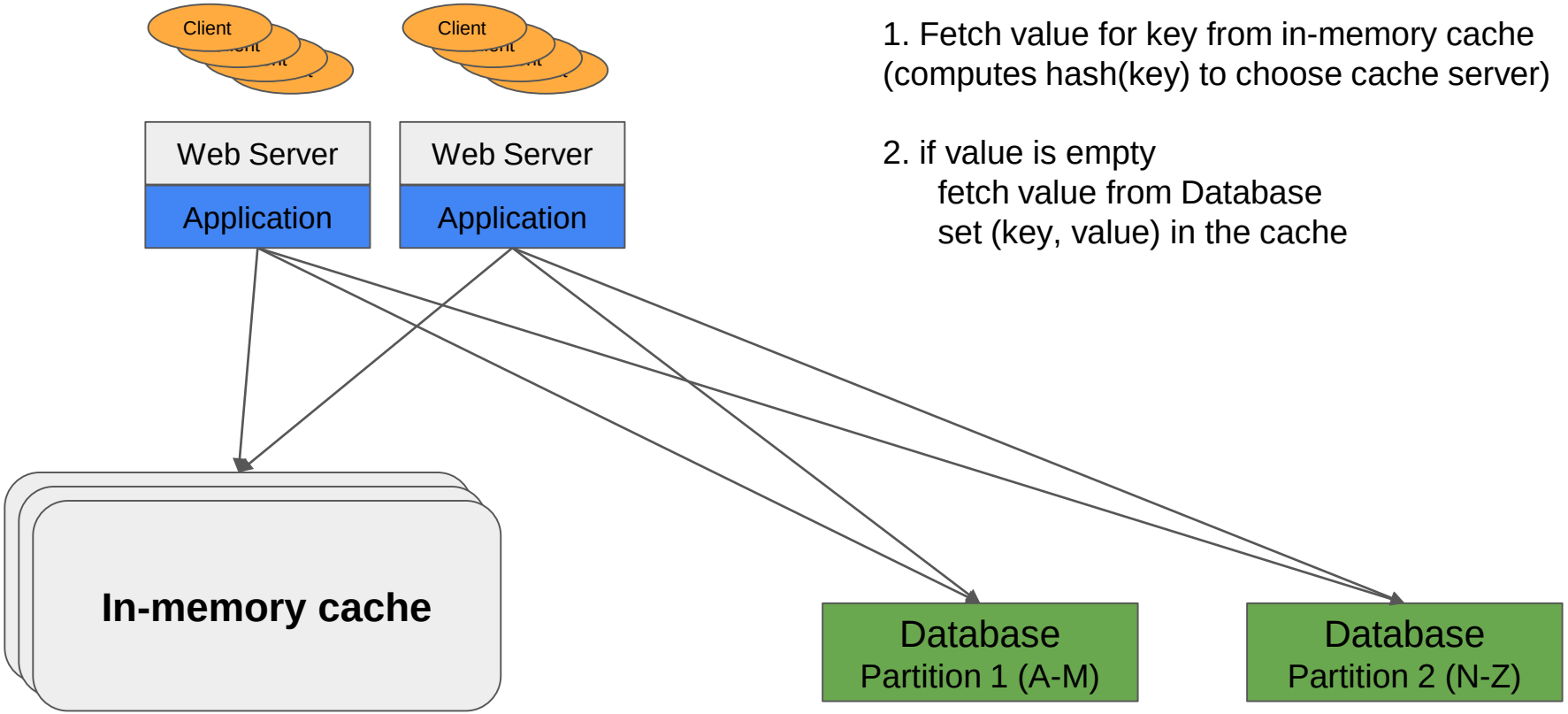
Introduce an in-memory cache



Memcached


Client
Client
Client
Client
Client
Web Server
Application
Web Server
Application
Web Server
Application
Web Server
Application
Web Server
Application
In-memory cache
{k1: v1, k2: v2, k3: v3}
Database
Partition 1 (A-M)
Database
Partition 2 (N-Z)

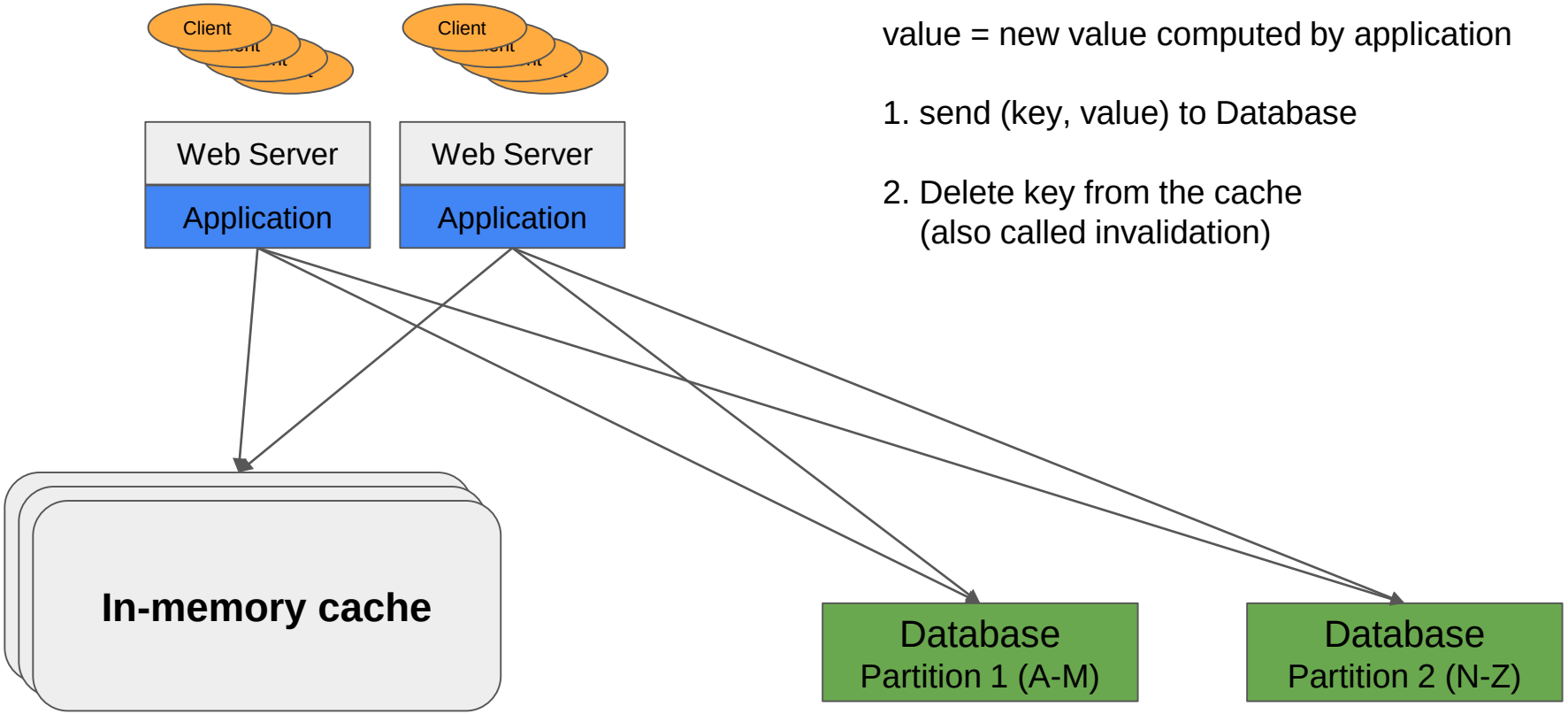
How Reads work?



1. Fetch value for key from in-memory cache (computes hash(key) to choose cache server)

2. if value is empty
   fetch value from Database
   set (key, value) in the cache

How Writes work?



value = new value computed by application

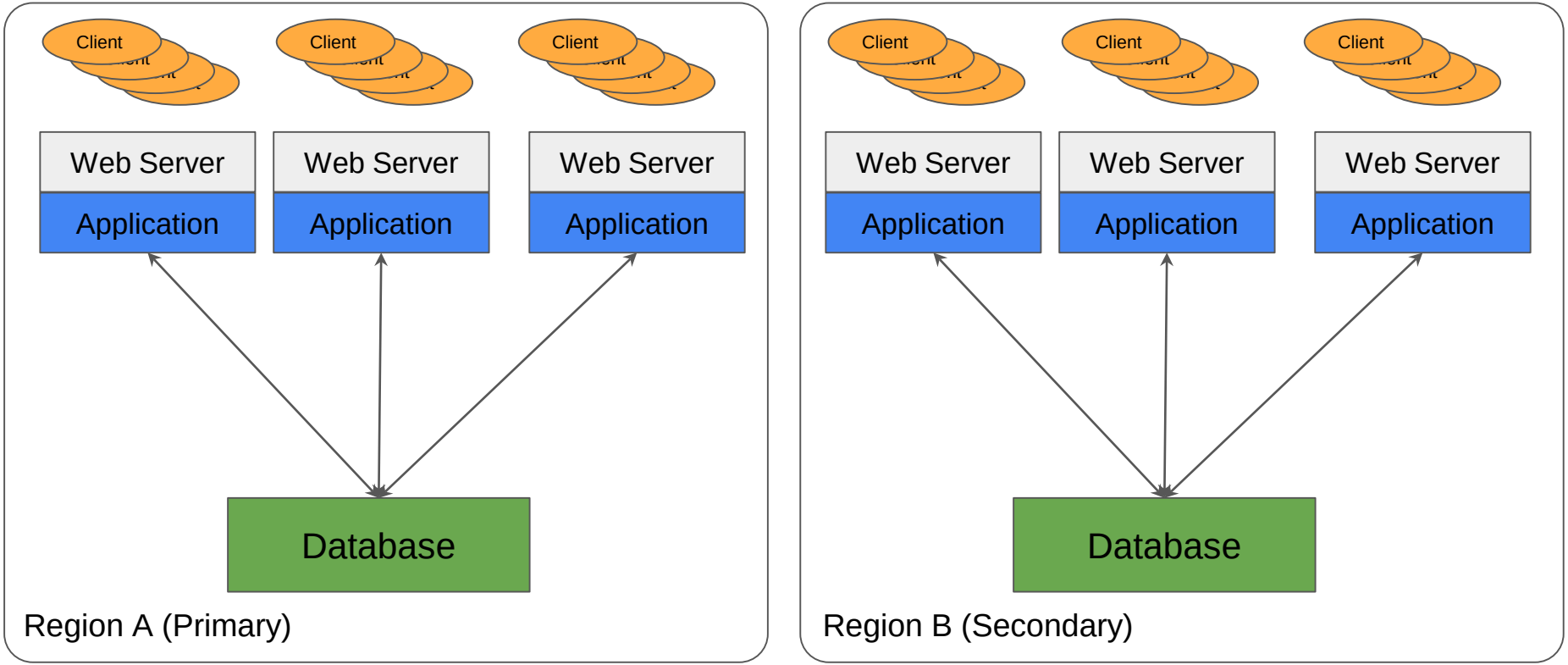
1. send (key, value) to Database

2. Delete key from the cache
   (also called invalidation)

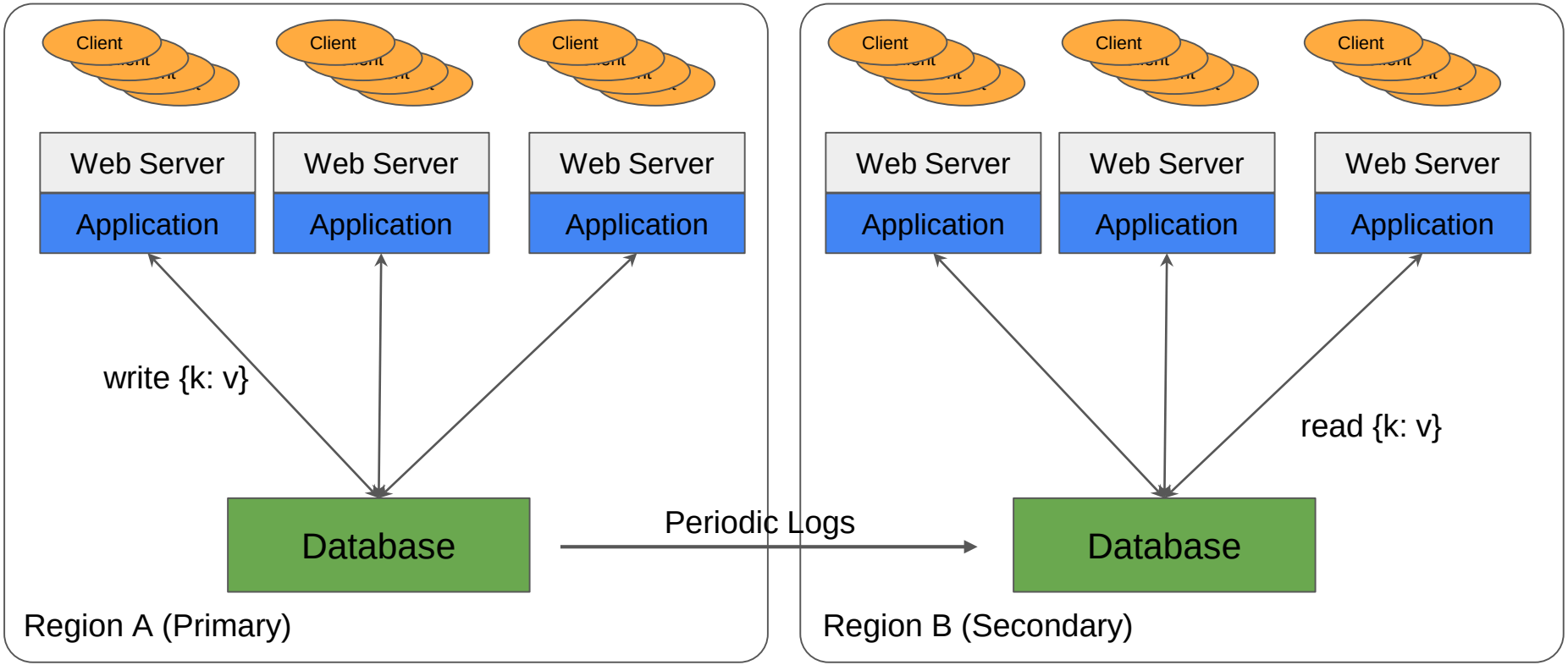
Why does this design work?

- Performance comes from the parallelism due to many servers.
- Many active users, means many web servers (clients).
- Caching layer absorbs most of the read requests, shielding the database.
- A very "popular" key can further be cached on multiple servers.

What about replication?



All writes operations go to the primary region.
Once they are written to primary database, they are streamed to secondary.



Implications of this choice

- Writes go direct to the primary database, are serialized with transactions.
  - So Writes are consistent.
- However, Reads do not always see the latest Write.
- A reasonable choice for Facebook, because data consistency isn't critical.
  - Updates will be seen "within a few seconds".
- Called eventual consistency in formal terms.
- However, a user will "read-your-own-writes" (because of delete())
  - Probably a good enough model for Facebook's use-case.

Lessons from this paper

- Look-aside caching is more challenging than it looks
  - Maintaining consistency in presence of multiple caches and replicas is hard.
- Caches are critical
  - Not only for reducing user-visible delay.
  - But also shielding database from huge overload.
- Stale reads are potentially a big headache
  - Need to avoid unbounded staleness (e.g. missing a delete() entirely)
  - Prefer to read-your-own-writes, for a saner consistency.
  - But more caches means more sources of staleness.
- Huge "fan-out" => parallel fetching, incast congestion.

Outline

- Architecture of a Distributed Application
- Memcached at Facebook
- Spanner at Google
- Takeaways

Google Spanner : Motivation

- Google F1 advertising database
- Was previously partitioned over many MySQL and BigTable DBs
- Needed better (synchronous) replication.
- More flexible sharding.
- Cross-shard transactions.
- Workload was dominated by read-only transactions.
- Strong consistency was required.

Motivating Example

- Distributed Transaction

```
BEGIN
    X = X - 100;
    Y = Y + 100;
END
```

We don't want any read or write of x or y sneaking between the two operations. After commit, all reads should see our updated X and Y.

- Why transactions?
  - Provide data consistency, even in presence of failures.
  - Make it easier for developers to reason about invariants.

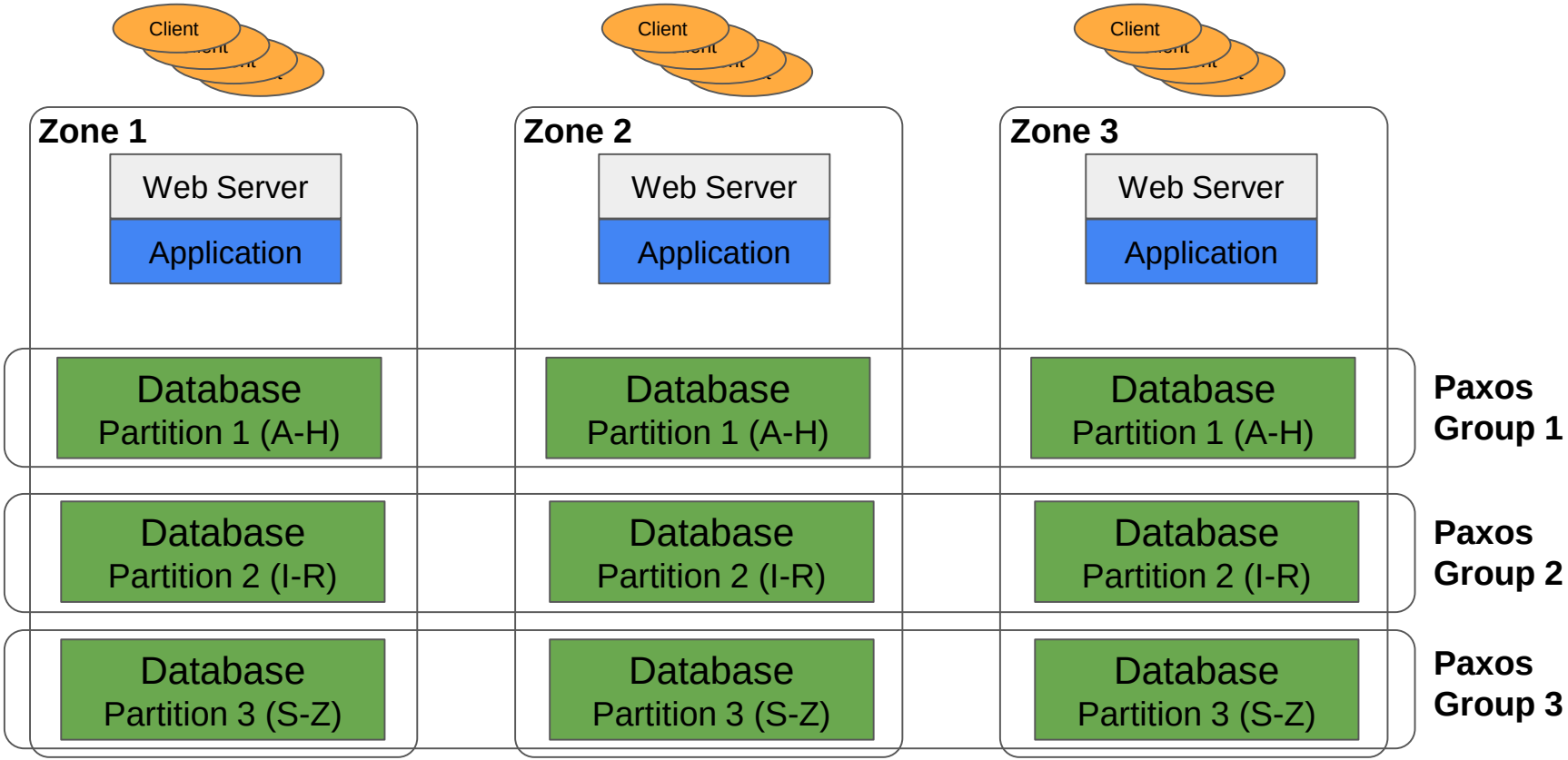
Google Spanner

One of the first systems to provide distributed transactions over the wide-area.

Key ideas:

- Two-phase commit over Paxos.
  - Before this two-phase commit was viewed as too slow and prone to blocking.
- Synchronized time for fast read-only transactions.
  - Using GPA and atomic clocks inside data centers!

Basic Terminology and Setup



Why such a setup?

- Sharding (partitions) allows huge total throughput via parallelism.
- Data centers can fail independently, and still make progress.
- Clients can read local replicas, fast, low latency!
- Can place replicas near relevant customers.
- Paxos requires only a majority, can tolerate slow/distant replicas.

Challenges

- Reads from a local replica must yield fresh data.
- But a local replica may not reflect latest Paxos writes.
- A transaction may involve multiple shards (multiple Paxos groups).
- Transactions that read multiple records must be serializable.
- But local shards may reflect different subsets of committed transactions.

How does Spanner achieve all this?

- It separates Read-Write Transactions from Read-Only transactions.
- An uses tightly synchronized clocks in a novel way.

Example of a Spanner Transaction

```
BEGIN
  read A -> acquire read lock on A
  read B -> acquire read lock on B
  write A -> promote A's lock to write lock
COMMIT (perform two-phase-commit)
  Coordinator -> A, B: prepare
  A, B -> OK
  Coordinator -> A, B: commit
```

A and B could be stored in different shards with different Paxos leaders.

Read-Write Transactions

Spanner does two-phase commit (2PC), with Paxos-replicated participants.

1. On Begin, client picks a unique Transaction ID (TID).
2. Client sends each Read to the Paxos leader of the relevant shard.
    a. Each shard first acquires a read lock on the relevant record.
3. Client keeps writes private until commit.

Read-Write Transactions (Commit procedure)

When a client commits,

1. Chooses a Paxos group to act as 2PC Transaction Coordinator (TC).
    a. Sends writes to relevant shard leaders.
2. Shard leaders acquire lock(s) on the written record(s).
    a. Log a "prepare" record via Paxos, to replicate lock and new value.
    b. Tell TC it is prepared (or not).
3. TC decides commit or abort.
    a. Logs the decision to its group via Paxos.
    b. Tell participant leaders and client the result.
4. Participant leaders log the TC's decision via Paxos.
    a. Release the transaction's locks.

Design implications

- Locking (two-phase locking) ensures serializability.
- 2PC was widely hated because it blocks with locks held if TC fails.
  - Replicating the TC with Paxos solves this problem.
- Read/Write transactions are expensive! and take a long time.
  - Many inter-data-center messages.
- However, lots of parallelism: many clients and many shards.
  - So total throughput could be high if not many conflicts.

Where do synchronized clocks come in?

These are required to enable lock-free read only transactions.

And provide external consistency (similar to strict serializability).

Serializability ensures that executing concurrent transactions is equivalent to executing the transactions serially in some order.

Strict serializability enforces that the order must be real-time order, i.e. if a transaction t2 starts after t1 commits, it must see t1's updates.

TrueTime

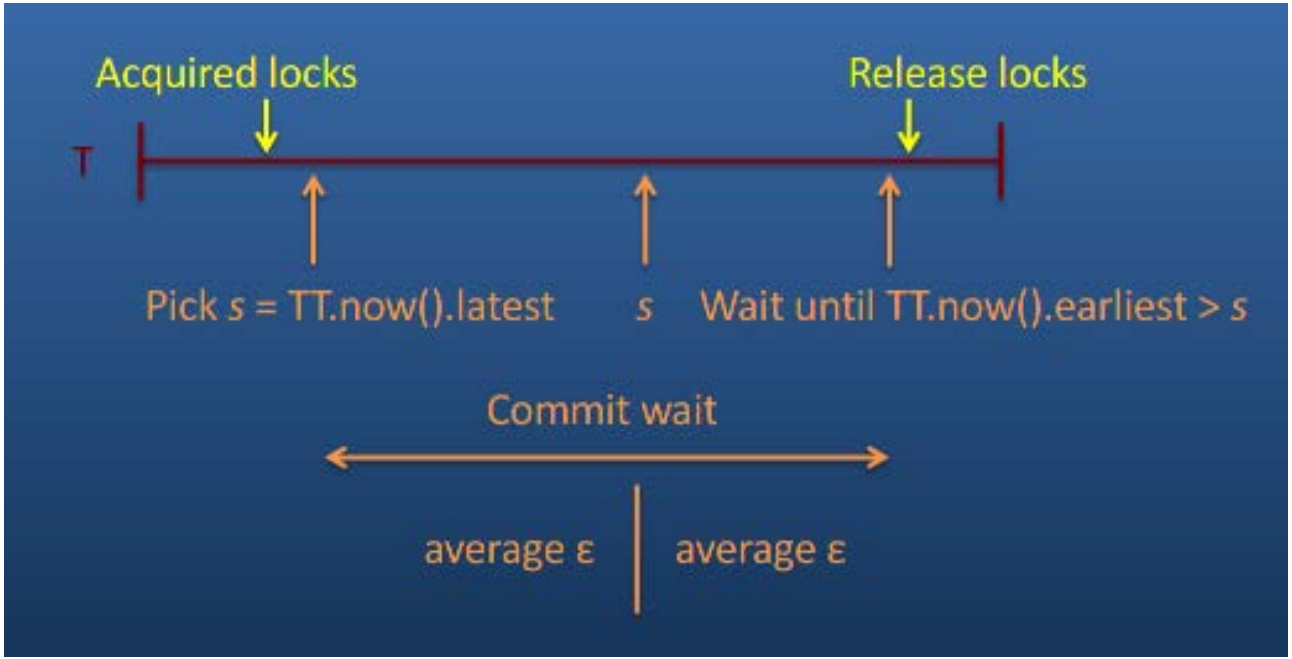
- Synchronized clocks across Google’s entire infrastructure
- Time master: a time server; set per data center
  - GPS receivers with dedicated antenna, or atomic clock
- Timeslave daemon: a time client on a workstation
  - Marzullo’s algorithm to sync clock and detect/reject liars
- Interface
  - TTInterval tt = TT.now();
  - tt.latest – tt.earliest = e, the instantaneous error bound
  - In practice, e sawtooths between 1ms and 6ms

How is TrueTime utilized?

Used to assign a commit timestamp to each transaction, in the order they were committed.

- The 2PC transaction coordinator gathers several TrueTime timestamps
  - The prepare timestamps from all participants.
  - Timestamp in the commit message from the client.
- TC picks an overall transaction timestamp to be greater than any of the above timestamps, and greater than any timestamp assigned to earlier transactions.
- TC then waits for this time to elapse, i.e. TTnow.earliest is past this time.

Commit Wait


Acquired locks
Release locks
T
Pick s = TT.now().latest
s
Wait until TT.now().earliest > s
Commit wait
average ε
average ε

Implications of TrueTime

- The larger the uncertainty bound from TrueTime, the longer commit wait period.
- Commit wait will slow down dependent transactions, since locks are held during commit wait.
- So, as time gets less certain, Spanner gets slower.
- Attack Vector: you can cause very long commit wait periods – slow the system down by messing with the clock.

Read-Only Transactions

Spanner eliminates two big costs for read-only transactions:

1. Read from local replicas, to avoid Paxos and cross-datacenter messages
    - But note, a local replica may not be up to date.
2. No locks, no two-phase commit, no transaction manager.
    - Again to avoid cross-data center messages to Paxos leader.
    - And to avoid slowing down read-write transactions.

Spanner does this by keeping multiple versions and "Snapshot Isolation".

Read-Only Transactions

Read-only transactions simply read the latest committed values.

How do we know the latest committed values?

- Using TrueTime!
- TC selects a time TTnow.latest(), which ensures are committed read/write transaction updates are available.
- TC performs the reads at those timestamps using the multiple versions.

Spanner Performance

replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

Table 3: Operation microbenchmarks. Mean and standard deviation over 10 runs. 1D means one replica with commit wait disabled.

- Latency of read-only transaction is 10x lower than that of read-write transactions.

Spanner after-thoughts

- Pretty rare to see a deployed system that offers distributed transactions over geographically distributed data.
- Spanner was a surprising demonstration that it can be practical.
- Timestamping scheme was the most interesting aspect.
- Widely used within Google; a commercial Google service; very influential.

Facebook Memcache

- Writes must go to the primary site's MySQL.
- Non-primary sites cannot write on their own, can have inconsistencies.
- But reads are blindingly fast (1,000,000 per second per memcache server).

Google Spanner

- Writes involve Paxos and perhaps two-phase commit.
- Paxos quorum for write must wait for some remote sites.
- No single site can write on its own.
- But has read transactions, consistent and fairly fast.

Other challenging problems

How to load-balance all clients among FrontEnd web servers?