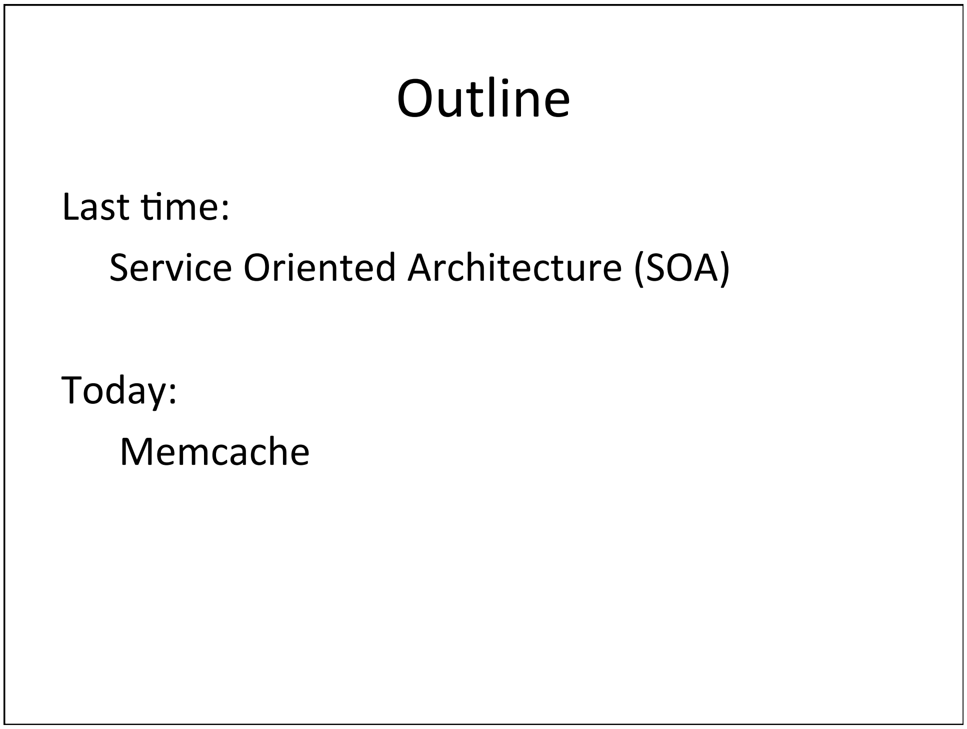
Memcache

Tom Anderson



Outline

Last time:

Service Oriented Architecture (SOA)

Today:

Memcache

Facebook’s Scaling Problem

- Rapidly increasing user base
  - Small initial user base
  - 2x every 9 months
  - 2013: 1B users globally
- Users read/update many times per day
  - Increasingly intensive app logic per user
  - 2x I/O every 4-6 months
- Infrastructure has to keep pace

Goals

Scale

- Bzillions of users
- Every user on FB all the time

Performance

- Low latency for every user everywhere

Fault tolerance

- Scale implies failures

Consistency model:

- “Best effort eventual consistency”

Strategy

Adapt off the shelf components where possible

Application logic needs to support rapid change

Speed of adding new features >> efficient operation

Support third party apps (SOA)

Fix as you go

- no overarching plan

Rule of thumb from the growth of the Internet:

- Every order of magnitude requires a rethink

Scaling

- A few servers
- Many servers
- An entire data center
- Many data centers

Each step 10-100x previous one

Workload

Each user’s page is unique

- draws on events posted by other users

Users not in cliques

- For the most part

User popularity is zipf

- Some user posts affect very large #’s of other pages
- Most affect a much smaller number

Question

Will clustering users be likely to work?

Workload

- Many small lookups
- Many dependencies
- Low spatial locality: all to all
- App logic: many diffuse, chained reads
  - latency of each read is crucial
- Much smaller update rate
  - still large in absolute terms

Data Center Network

- Data center capacity is non-uniform
  - Oversubscribed folded Clos built out of switches with 10-40 ports
  - Maintaining locality is important

Facebook Three Layer Architecture

- Application front end
  - Stateless, rapidly changing program logic
  - If app server fails, redirect client to new app server
- Memcache
  - Lookaside key-value cache
  - Keys defined by app logic
- Fault tolerant storage backend
  - Stateful
  - Careful engineering to provide safety and performance
  - Both SQL and NoSQL

Scale By Hashing: Shards

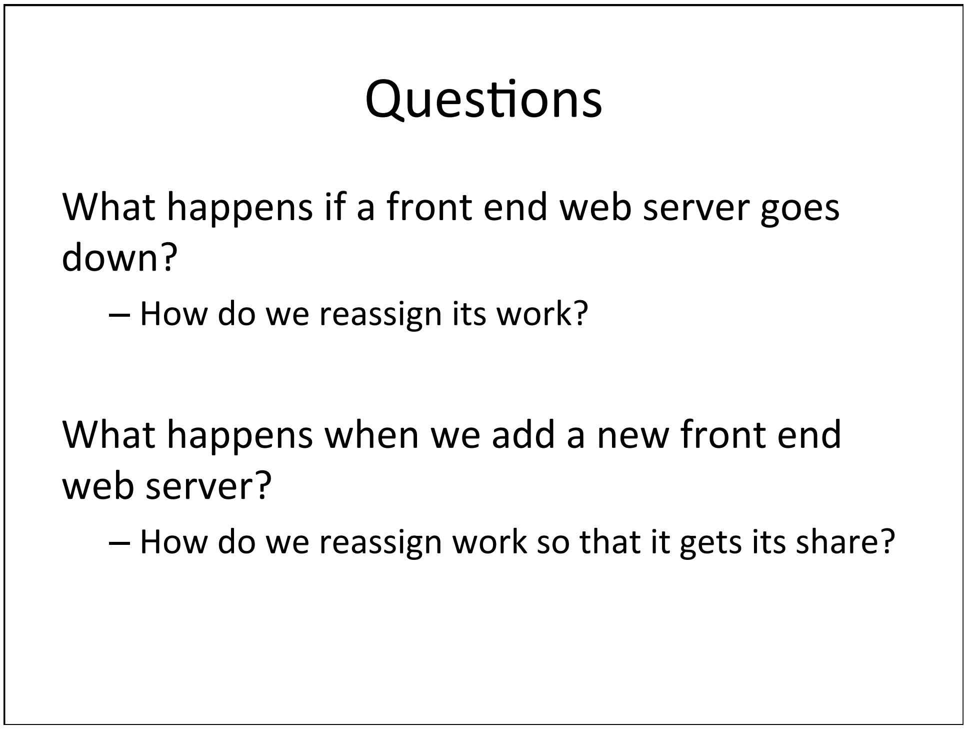
Hash users to front end web servers

Hash keys to memcache servers

Hash files to SQL servers

App code is all to all

- a given user will pull data from a large # of memcache and storage servers



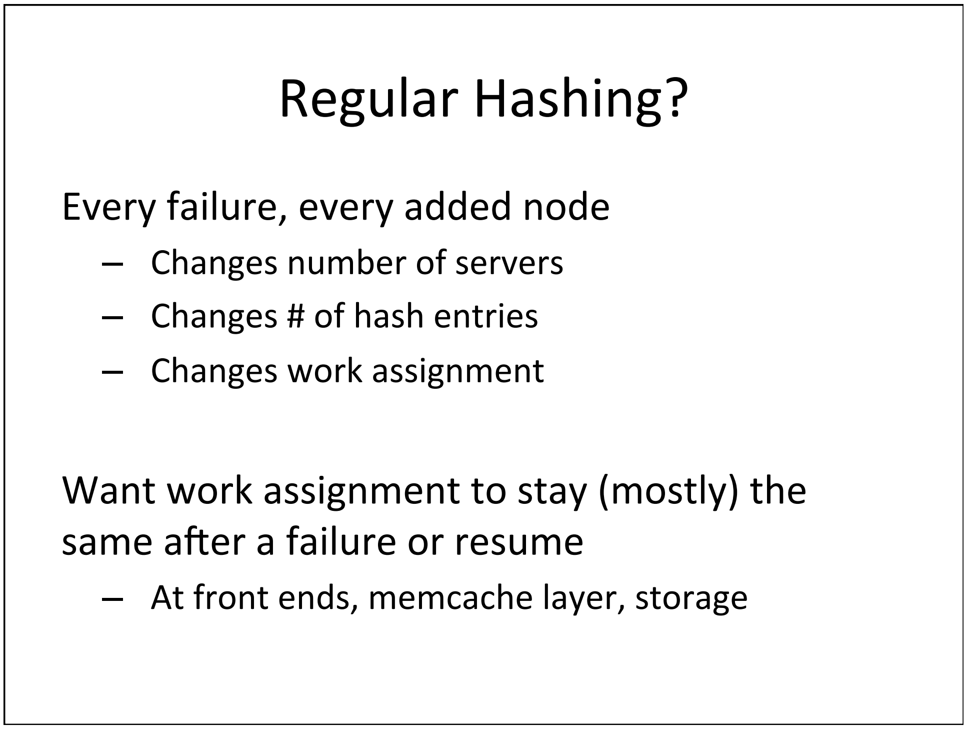
Questions

What happens if a front end web server goes down?

– How do we reassign its work?

What happens when we add a new front end web server?

– How do we reassign work so that it gets its share?



Regular Hashing?

Every failure, every added node

- Changes number of servers
- Changes # of hash entries
- Changes work assignment

Want work assignment to stay (mostly) the same after a failure or resume

- At front ends, memcache layer, storage

Consistent Hashing

Hash clients/keys and servers onto the same ID space

Sort all the servers by their hash value H(Si) < H(Sj)

– Renumber so … < H(Si-1) < H(Si) < H(Si+1) < …

Server Si's workload:

All clients/keys, st H(Si) < H(key) < H(Si+1)

Questions

How unbalanced is regular hashing, on average?

How unbalanced is consistent hashing?

If workload is uniform random?

If workload is zipf?

Consistent Hashing Fault Tolerance

If Si fails, assign its keys to server Si-1

- How does load balance change when remove a node?

If new Sj hashes to value between Si, Si+1: assign it keys between H(Sj), H(Si+1)

- How does load balance change when add a node?

Consistent Hashing Optimization

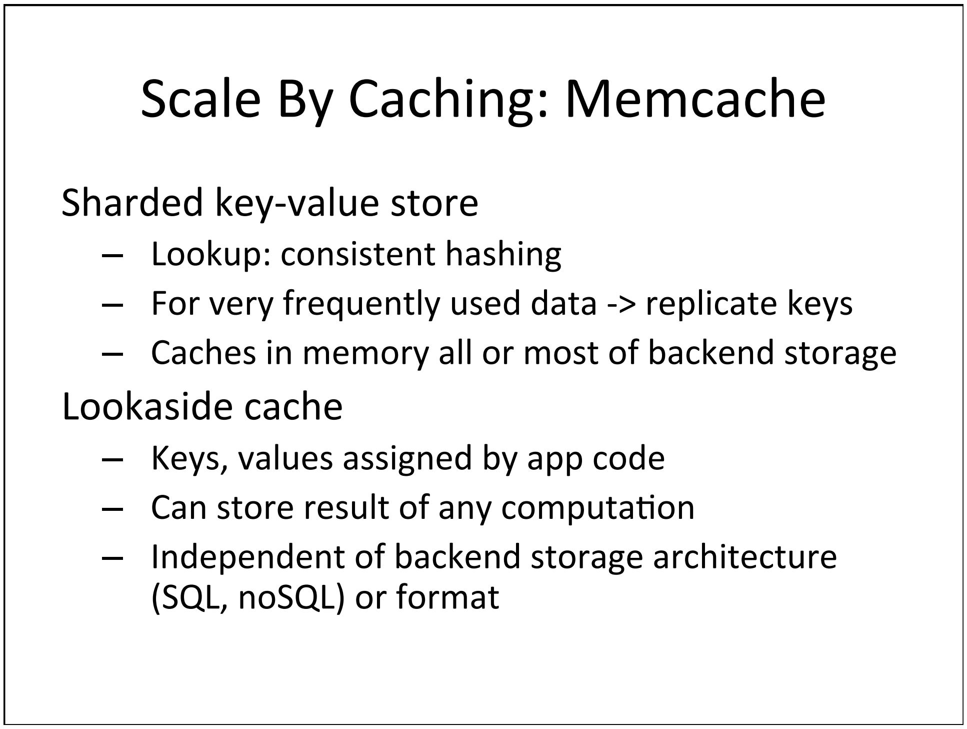
Create 100 “virtual servers” for each server

Assign keys based on hash of virtual server ID

Reduces load imbalance by ~10x

Speeds reconfiguration after a failure

- Workload for each “failed” virtual node spread to a different peer



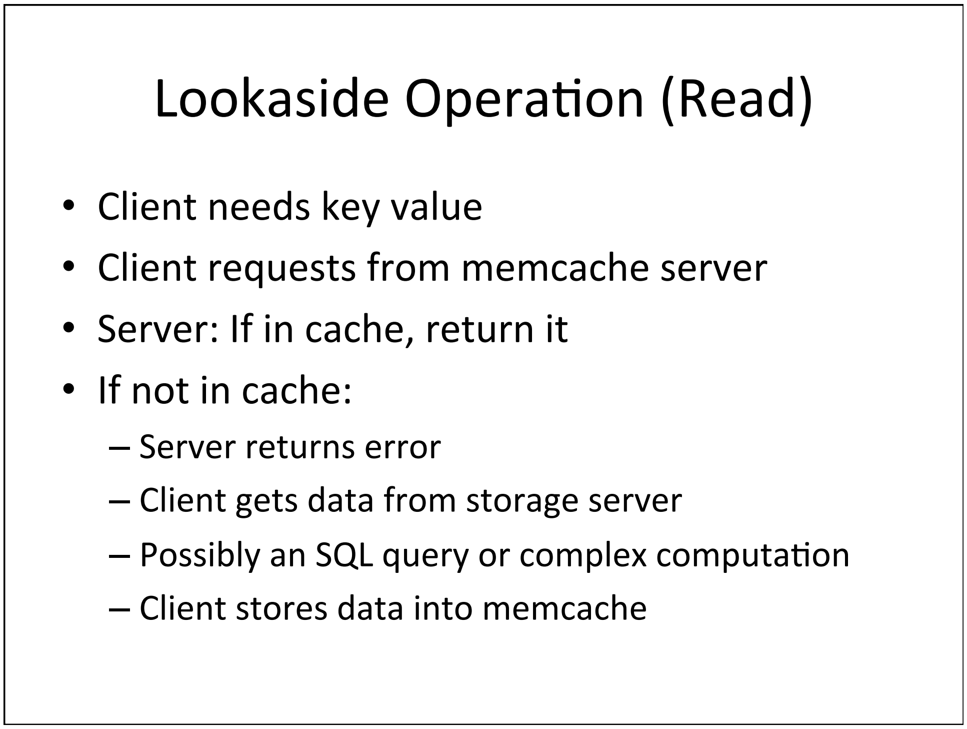
Scale By Caching: Memcache

Sharded key-value store

- Lookup: consistent hashing
- For very frequently used data -> replicate keys
- Caches in memory all or most of backend storage

Lookaside cache

- Keys, values assigned by app code
- Can store result of any computation
- Independent of backend storage architecture (SQL, noSQL) or format



Lookaside Operation (Read)

- Client needs key value
- Client requests from memcache server
- Server: If in cache, return it
- If not in cache:
  - Server returns error
  - Client gets data from storage server
  - Possibly an SQL query or complex computation
  - Client stores data into memcache

Question

What if swarm of users read same key at the same time?

Lookaside Operation (Write)

- Client changes a value that would invalidate a memcache entry
  - Could be an update to a key
  - Could be an update to a table
  - Could be an update to a value used to derive some key value
- Client puts new data on storage server
- Client invalidates entry in memcache

Memcache Consistency

Is memcache linearizable?

Example

Thread A: Reader

```
Read cache
If missing,
  Fetch from database
  Store back to cache
```

Thread B: Writer

```
Change database
Delete cache entry
```

Interleave any # of readers/writers

Example

Thread A: Reader	Thread B: Writer
	Change database
Read cache	
	Delete cache entry

Memcache Consistency

What if we delete cache entry, then change database?

Example

Thread A: Reader	Thread B: Writer
	Delete cache entry
Read cache	
Fetch data from database	
	Change database
Store fetched data to memcache	

Memcache Consistency

Is memcache linearizable considering only the gets/puts to a single key?

Example

- A: Read cache
- A: Read database

- B: change database
- B: Delete entry

- A: Store back to cache

Lookaside With Leases

Goals:

- Reduce (eliminate?) per-key inconsistencies
- Reduce cache miss swarms

On a read miss:

- leave a marker in the cache (fetch in progress)
- return timestamp
- check timestamp when filling the cache
- if changed means value has (likely) changed: don't overwrite

If another thread read misses:

- find marker and wait for update (retry later)

Question

What if web server crashes while holding lease?

Question

Is Facebook lookaside with leases linearizable for operations to a single key?