Handling Load at Scale

- Motivation for partitioning state:
  - Single server incapable of handling load
  - More shards $\rightarrow$ Lesser load per shard

- Partitioning beneficial only up to a limit
  - More fine-grained shards $\rightarrow$ Each request will touch more shards

- Complementary strategy: Caching
Scaling Memcache at Facebook

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

- Demand-filled cache: Populate cache when handling cache misses
- Contrast: Google Ads uses proactive caching
Write-Through Cache

- Read from cache, which reads from DB upon miss
- Both DB and cache updated before write complete
- Cache will become a bottleneck
Write-Through Cache

- Replicate cache to balance load
- How to keep all cache copies consistent with DB?
- DB needs to invalidate/update other cache copies
- Challenging due to heterogeneity of data stores
Look-Aside Cache
Write Protocol

- First update, then delete
  - Temporary inconsistency okay

- Why delete, not update?

- Race condition:
  - C1 updates DB
  - C2 updates DB
  - C2 updates cache
  - C1 updates cache
Persistent Inconsistency

Race condition between concurrent reader and writer resulting in permanently stale cached value? Fix?

- C1 get(k) misses, reads v1 from DB
- C2 updates DB to v2, deletes k from cache
- C1 sets(k, v1) in cache
Client Leases

- When get on cache misses, client is granted lease
  - Client has lock on updating cache while lease valid
- Lease invalidated by cache delete
- Leases also help cope with “thundering herds”
Incorporating Cold Caches

- Clients partitioned across caches to balance load
- When new cache added, all gets on it will miss
- Spike in load on DB
Incorporating Cold Caches

- Clients assigned to cold cache
  - Upon cache miss, try get on warm cache before DB
  - Set and delete on cold cache

- Race condition between writer and reader?
Incorporating Cold Caches

- Clients assigned to cold cache
  - Upon cache miss, try get on warm cache before DB
  - Set and delete on cold cache

- Race condition?
  - C1 updates k to v2 in DB, deletes k in cold cache
  - C2’s get results in cache miss on cold cache, reads v1 from warm cache, sets v1 in cold cache

- Fix: After delete on cold cache, sets disabled for two seconds
  - Within that period, DB invalidates warm cache
Other Perf. Optimizations

- Vary transport protocol based on request type
  - UDP for gets
  - TCP for sets and deletes
- Batch gets
Announcements

- Reminder: Project 4 due on Thursday
  - Write linearizability checker for part A
  - Ensure design for part C prevents deadlocks

- Sample final exam posted
  - Solutions will be discussed in class next Tuesday

- Submit final teaching evaluations
Fault Tolerance of 2PL

More shards $\rightarrow$ Greater chance that one shard unavailable
Impact of multi-partition operations on user-perceived latency:
- Greater the # of shards that a transaction touches, higher the latency

Why?
- Transaction latency = max(per-request latency)
- Transaction slow if response from any one shard is slow
Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO
Impact of Tail Latency

The graph illustrates the probability of service latency exceeding 1s as a function of the number of servers for different failure rates:

- 1 in 100
- 1 in 1,000
- 1 in 10,000

Key points:
- At 1 server, the probability for each curve is approximately 0.63.
- At 2,000 servers, the probability is approximately 0.18 for 1 in 10,000 failures.
### Data from a service at Google

- Root server receives request from user and executes request at many leaf servers.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>50%ile latency</th>
<th>95%ile latency</th>
<th>99%ile latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>One random leaf finishes</td>
<td>1ms</td>
<td>5ms</td>
<td>10ms</td>
</tr>
<tr>
<td>95% of all leaf requests finish</td>
<td>12ms</td>
<td>32ms</td>
<td>70ms</td>
</tr>
<tr>
<td>100% of all leaf requests finish</td>
<td>40ms</td>
<td>87ms</td>
<td>140ms</td>
</tr>
</tbody>
</table>

- **t = 0**: Requests issued
- **t = 10ms**: 50% responses received
- **t = 70ms**: 95% responses received
- **t = 140ms**: 100% responses received
Causes for Tail Latency

- Why might a server be occasionally slow in responding to a request?
  - Infrastructure shared by services
  - Background work
  - Energy management
Solution: Add Redundancy

- Exploit fact that every server’s state is replicated

- When sending request to a server, concurrently send requests to replicas
  - Take first response

- Problem?
  - Increased load will worsen latencies
Efficient Use of Redundancy

- **Option 1:**
  - Issue request first to any one replica
  - Issue requests to other replicas after timeout
  - Increase in load only when first response is slow
  - Tradeoff between timeout and load

- **Option 2:**
  - Issue requests to all replicas almost simultaneously
  - Tell every replica to cancel request at other replicas when it responds
Other Solutions for Tail Latency

- Selectively increase replication for hot partitions
- Detect and put slow machines on probation
- Tradeoff quality of response for latency
  - Examples?
    - Google search, Facebook news feed
Next time ...

- Bitcoin
- Research at Michigan