Memcache

Tom Anderson

Outline

Last time:
  Consistent hashing, Memcache intro

Today:
  Memcache
Facebook

• Scale by hashing to partitioned servers
• Scale by caching
• Scale by replicating popular keys
• Scale by replicating clusters
• Scale by replicating data centers

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

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
### Example

<table>
<thead>
<tr>
<th>Thread A: Reader</th>
<th>Thread B: Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read cache</td>
<td>Change database</td>
</tr>
<tr>
<td>If missing,</td>
<td>Delete cache entry</td>
</tr>
<tr>
<td>Fetch from database</td>
<td></td>
</tr>
<tr>
<td>Store back to cache</td>
<td></td>
</tr>
</tbody>
</table>

Interleave any # of readers/writers

### Example

<table>
<thead>
<tr>
<th>Thread A: Reader</th>
<th>Thread B: Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read cache</td>
<td>Change database</td>
</tr>
</tbody>
</table>

Delete cache entry
Memcache Consistency

Is the lookaside protocol eventually consistent?

Example

A: Read cache, miss
A: Read database

B: change database
B: Delete memcache 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 timestamp changed => value (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 lookaside with leases linearizable?

Example

Thread A: Reader

Thread B: Writer

Change database

Read cache

Delete cache entry
Question

Is this eventually consistent?

Example

Thread A: Reader
Read cache

Thread B: Writer
Change database

CRASH!
(before Delete cache entry)
Question

Linearizable?
– read misses obtain lease
– writes obtain lease (prevent reads during update)

Except that
– FB replicates popular keys (need lease on each copy?)
– FB bypasses the cache on pkt loss
– memcache server might fail, or appear to fail by being slow (e.g., to some nodes, but not others)

Latency Optimizations

Concurrent lookups
– Issue many lookups concurrently
– Prioritize those that have chained dependencies

Batching
– Batch multiple requests (e.g., for different end users) to the same memcache server

Incast control:
– Limit concurrency to avoid collisions among RPC responses
More Optimizations

Return stale data to web server if lease is held
  – No guarantee that concurrent requests returning stale data will be consistent with each other

Partitioned memory pools
  – Infrequently accessed, expensive to recompute
  – Frequently accessed, cheap to recompute
  – If mixed, frequent accesses will evict all others

Key replication when access rate is too high for one server

Gutter Cache

When a memcache server fails, flood of requests to fetch data from storage layer
  – Slows users needing any key on failed server
  – Slows other users due to storage server contention

Solution: backup (gutter) cache
  – Time-to-live invalidation (ok if clients disagree as to whether memcache server is still alive)
  – Backup cache also suggested in Yegge
Scaling Within a Cluster

What happens as we increase the number of memcache servers to handle more load?

- Batching less effective
- More replication of popular servers
- More failures hit gutter cache
- ...

Multi-Cluster Scaling

Multiple independent clusters within data center

- Each with front-ends, memcache servers
- Data replicated in the caches in each partition
- Shared storage backend

Web server driven invalidation?

- need to invalidate every cluster on every update

Instead: mcsqueal
mcsqueal

Web servers talk to local memcache. On update:
- Acquire local lease
- Tell storage layer which keys to invalidate
- Update local memcache

Storage layer sends invalidations to other clusters
- Scan database log for updates/invalidations
- Batch invalidations to each cluster (mcrouter)
- Forward/batch invalidations to remote memcache servers

Per-Cluster vs. Multi-Cluster

Per-cluster pools of memcache servers
- Frequently accessed data
- Inexpensive to compute data
- Lower latency, less efficient use of memory

Shared multi-cluster pools
- Infrequently accessed
- Hard to compute data
- Higher bandwidth on oversubscribed clos network
Cold Start Consistency

During new cluster startup, on cache miss:
- Web frontend checks remote memcache cluster for data
- Puts fetched data into local pool
- Subsequent requests fetch from local pool

Example

A: Local cache miss
A: Read remote cluster
A: Put data in local cache

B: change database
B: queue remote invalidation
B: Delete memcache entry

Apply remote invalidation

Solution: prevent cache fills within 2 seconds of delete
Multi-Region Scaling

Storage layer consistency
- Storage at one data center designated as primary
- All updates applied at primary
- Updates propagated to other data centers
- Invalidations to memcache layer at delayed until after update reaches that site

However
- Frontends may read stale data
- Even data that they just wrote

Multi-Region Consistency

To perform an update to key:
- put marker into local region
- Send write to primary region
- Delete local copy

On a cache miss:
- Check if local marker
- If so, fetch data from primary region
- Fill local copy
Data Centers without Data

Tradeoff in increasing number of data centers
- Lower latency when data near clients
- More consistency overhead
- More opportunity for inconsistency

Mini-data centers
- Front end web servers
- Memcache servers
- No backend storage: remote access for cache misses