Scalable Data Storage at Facebook

Ashvin Goel
ECE454, University of Toronto

Slides are modified from the original talks by Rajesh Nishtala and Nathan Bronson
Requirements at Facebook

- Near real-time communication
- Aggregate content on-the-fly from multiple sources
- Be able to access and update popular shared content
- Scale to process millions of user requests per second
- Support heavy read load (over 1 billion reads/sec)
- Scale to petabytes of storage
- Geographically distributed users
Dynamically Rendering the Social Graph
Storing the Social Graph

mysql databases

Web Server (PHP)
Objects and Associations

- Objects, associations stored in separate database tables
  - Objects identified by unique 64-bit IDs
    - Each object has a type field and other data fields
  - Associations identified by <id1, type, id2>
  - Associations have a time field and other data fields

id: 1807 =>
  type: POST
  str: "At the summ...

id: 308 =>
  type: USER
  name: "Alice"

id: 2003 =>
  type: COMMENT
  str: "how was it ..."
Association Lists

- Association queries often require returning a list of associations:
  - assoc_get(id1, type) returns <id1, type, *>, ordered by time

```
id: 1807 => type: POST
  str: “At the summ..."

<1807,COMMENT,4141>
  time: 1,371,709,009
  str: “Been wanting to do ..."

<1807,COMMENT,8332>
  time: 1,371,708,678
  str: “The rock is flawless, ..."

<1807,COMMENT,2003>
  time: 1,371,707,355
  str: “how was it, was it w..."

id: 2003 => type: COMMENT

```
Early Days (Pre-Caching)

• Just a few databases were enough to support the load

• Data partitioned by object id across the databases
  • All outgoing associations for an object stored in same partition
  • Enables association queries to be served from one partition
Problem

- High fanout and multiple rounds of data fetching

Data dependency DAG for a small request
Scaling memcache in 4 “easy” steps

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No memcache servers (pre-memcache)</td>
</tr>
<tr>
<td>1</td>
<td>A few memcache servers</td>
</tr>
<tr>
<td>2</td>
<td>Many memcache servers in one cluster</td>
</tr>
<tr>
<td>3</td>
<td>Many memcache servers in multiple clusters within a region</td>
</tr>
<tr>
<td>4</td>
<td>Geographically distributed clusters in multiple regions</td>
</tr>
</tbody>
</table>
Facebook has two orders of magnitude more reads than writes (500:1)

Use a caching server called memcached

Cache Graph in Memcache
Caching Improves Read Performance

- Memcache is a single-node, key-value store that uses a hash table to store key and values
  - Store nodes as keys
  - Stores edge lists as values
- Reads performed from memcache
- Reduces load on database
Reading Data from Memcache

• Use memcache as a look-aside cache
  • Avoids any changes to memcache
• On read miss:
  • Lookup database
  • Set value in memcache
Handling Updates

- Memcache needs to be invalidated after DB write
- Prefer deletes to sets
  - Idempotent
  - Demand filled
- Up to web application to specify which keys to invalidate after database update
Stale Set Problem

1. Read (A)
Stale Set Problem

1. Read (A)

2. Another web server updates value to B
Stale Set Problem

1. Read (A)
2. Another web server updates value to B
3. Read (B)
4. Set (B)

Web Server A
Database B
Web Server B
Memcache B
Stale Set Problem

1. Read (A)
2. Another web server updates value to B
3. Read (B)
4. Set (B)
5. Set (A)

MC & DB Inconsistent
Avoiding Stale Set

Extend memcache protocol with “leases”
- On a read miss, return a lease-id (before Step 1)
- Lease-id is invalidated inside memcache on a delete (by Step 2)
- Disallow set if the lease-id is invalid at the server (at Step 5)

1. Read (A)
2. Another web server updates value to B
3. Read (B)
4. Set (B)
5. Set (A)
Thundering Herd Problem

Thundering Herd:
A key that is read and written heavily will cause many cache misses and accesses to the database.
Avoiding Thundering Herds

Memcache server returns leases with a rate limit, so the first read has a miss, the rest of the reads have a cache hit.
Scaling memcache in 4 “easy” steps

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No memcache servers (pre-memcache)</td>
</tr>
<tr>
<td>1</td>
<td>A few memcache servers</td>
</tr>
<tr>
<td>2</td>
<td>Many memcache servers in one cluster</td>
</tr>
<tr>
<td>3</td>
<td>Many memcache servers in multiple clusters within a region</td>
</tr>
<tr>
<td>4</td>
<td>Geographically distributed clusters in multiple regions</td>
</tr>
</tbody>
</table>
Need even more read capacity

• Use multiple memcache servers
  • Items are distributed across memcache servers by using consistent hashing on the key
• All web servers talk to all memcache servers
  • Accessing 100s of memcache servers to process a user request is common
Problem: Incast Congestion

- Many simultaneous responses overwhelm networking resources at web server
- Solution: Limit the number of outstanding requests by web server to memcache with a sliding window
  - Larger windows result in more congestion
  - Smaller windows result in more round trips to the network
### Scaling memcache in 4 “easy” steps

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No memcache servers (pre-memcache)</td>
</tr>
<tr>
<td>1</td>
<td>A few memcache servers</td>
</tr>
<tr>
<td>2</td>
<td>Many memcache servers in one cluster</td>
</tr>
<tr>
<td>3</td>
<td>Many memcache servers in multiple clusters within a region</td>
</tr>
<tr>
<td>4</td>
<td>Geographically distributed clusters in multiple regions</td>
</tr>
</tbody>
</table>
Scaling Problem

All-to-all communication between web servers and memcache servers limits horizontal scaling
Solution: Use Multiple Clusters

- Each cluster caches data in its own memcache server
  - A web server only accesses the memcache servers in its cluster
  - Helps limit # of servers per cluster
- All the clusters are backed by a single database installation
Problem: Need Cache Consistency

- Same data is cached (i.e., replicated) in different clusters in the memcache servers
- Need to keep caches consistent
Solution: Use Weak Consistency

- Facebook uses a weak consistency model called eventual consistency
  - Reads return locally cached values, which may be stale for a short time
  - Replicas will eventually have the same value after receiving all updates
- Consistency mechanism
  - Invalidate cached data after database is updated
Invalidating Caches

- Use the database to track all updates that have committed
  - Issue corresponding deletes (invalidations) to all caches
  - Commit log is persistent, invalidations can be resent, and caches resynchronized, on failure
Scaling Invalidations

- Aggregate (batch) deletes in the memcache routers
- Reduces packet rate by 18x
- Each stage buffers deletes in case downstream component is down
Scaling memcache in 4 “easy” steps

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No memcache servers (pre-memcache)</td>
</tr>
<tr>
<td>1</td>
<td>A few memcache servers</td>
</tr>
<tr>
<td>2</td>
<td>Many memcache servers in one cluster</td>
</tr>
<tr>
<td>3</td>
<td>Many memcache servers in multiple clusters within a region</td>
</tr>
<tr>
<td>4</td>
<td>Geographically distributed clusters in multiple regions</td>
</tr>
</tbody>
</table>
Geographically Distributed Clusters

Each region has a separate database replica
Geographically Distributed Clusters

One region holds master database, rest are read-only copies
Geographically Distributed Clusters

All writes from any of the regions are sent to master

Use database’s replication mechanism to synchronize replicas

Database replication is performed asynchronously, so replica databases may lag behind master
Write at Master Region

Reuse the database-based invalidation mechanism to ensure consistency of replica DB and memcache at replica region.

1. Write to master

3. Invalidation
Write at Replica Region

Problem: race between 1) Replication to Replica DB, 2) Reads from Replica DB

1. Write to master
2. Delete key in memcache
3. On miss, read stale value from DB
4. Set potentially state value in memcache
Remote Markers

Set a special flag in memcache that indicates that Replica db has stale value

- Read miss path:
  - If marker set
    - read from master DB
  - else
    - read from replica DB
Putting it All Together

- Start with a single front-end cluster
  - Caches store data partition
- Add multiple front-end clusters in region
  - Allows scaling by replicating caches
- Multiple Regions
  - Allows scaling by replicating databases
Rethinking Caching at Facebook

- Memcached based design works well but not ideal
  - Storing edge lists as values is inefficient
    - Retrieving or updating a single edge (association) requires retrieving the entire edge list
    - Limited support for time-based association queries

- Distributed cache consistency logic
  - Cache logic is run on web servers, not in caching server
  - Makes it hard to ensure cache consistency, especially during failures

- Memcached is not aware of graph semantics
  - Ensuring consistency is inefficient
Tao: Facebook’s Caching Layer

- Tao replaces memcached as a graph-aware caching layer

- Tao provides better support for object, association and association list queries

- Tao mediates all accesses to the database, and uses a two-level caching scheme to provide stronger cache consistency guarantees

- Tao is graph aware, allowing it to implement cache consistency more efficiently
Memcache Design

Web servers

Distributed control logic

Caches

Database
Tao’s Two-Level Cache

Web servers

Follower cache

Leader cache

Database

Cluster

Single cache coordinator per database
Graph-Aware Caching

Web servers

Follower cache

Leader cache

Database

X → Y

refill X

range get

Y, A, B, C

X → Y

ok

refill X

Y, A, B, C

X → Y

ok

Y, A, B, C

X → Y

ok

Y,...
Asynchronous DB Replication

Web servers

Follower cache

Leader cache

Database
Handling Various Challenges

- Failures can occur at each tier
  - The TAO paper describes how various failures are handled
- Load imbalance can occur across partitions in a tier
  - Partitions are cloned to balance load
- High degree objects with 1000s of associations
  - Need client-specific optimizations
Lessons Learned

• Push complexity into the client whenever possible
  • Simplifies design of caching and storage service
• Randomly partition the data (no locality)
  • Helps reduce hotspots
• Separate cache and persistent store
  • Allows them to be designed, scaled and operated independently
• Building the custom Tao cache is beneficial for Facebook’s demanding workload
• Reusing the standard MySQL database allows reusing standard asynchronous replication mechanisms, replica creation, bulk import, backup, monitoring tools, etc.
These slides are based on the following papers:

- Scaling Memcache at Facebook, NSDI 2013
- TAO: Facebook’s Distributed Data Store for the Social Graph, USENIX ATC 2013