Caching At Twitter and moving towards a persistent, in-memory key-value store

Manju Rajashekhar
@manju
Outline

Caching System Architecture

Twemcache

Twemproxy

Learnings

in-memory persistent store
Cache In Production

- ~30 TB of cache
- > 2000 instances of caches
- ~500 machines

Average cache instance is 15G

- ~2 trillion queries/day
- 23 million queries/sec
Cache Systems

Cache is an **Optimization for**

**CPU**

Disk (write through / write back)
Cache API

CRUD API (memcache)
- set("key", "value")
- get("key")
- delete("key")
- ...

DS API (redis)
- push("key", "element-1")
- pop("key")
- get("key", "index")
- ...

Twitter Inc. | @manju
Caching System: Components

Simple distributed components

- Client
- Proxy
- Server

- Routing / Sharding
- Heartbeating / Liveness
- Protocol Encoder / Decoder
- Object Store
Twemcache

Based on memcached 1.4.4

Running in production since Jan ’11

code: github.com/twitter/twemcache
Features

Custom Eviction Algorithm

Thread-local stats collector

Command Logger
Eviction (1)

LRU Eviction

New Item
Eviction (2)

Items of different sizes

New Item
Eviction (3)

Per Slabclass LRU Eviction = calcification, pseudo OOM
Slab Eviction

Slab Eviction = deterministic behavior
Motivation

Keys accessed/updated/retrieved in the past 24hrs
- What data is hot and what is not?
- What should the heap size be to cache for 24 hours worth of data?

How many times and when is a key retrieved/updated after insertion?
- Explains why hit rate is so
- Determine a reasonable TTL
- Helps construct a heat map to decide cache size / hit rate trade off

What’s the stats per namespace? (“foo:” vs “bar:”)
- Does co-habitat make sense?
Async Command Logger

Log Format

<table>
<thead>
<tr>
<th>Client IP</th>
<th>Timestamp</th>
<th>Type</th>
<th>Key</th>
<th>Status</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>172.25.135.205:55438</td>
<td>[09/Jul/2012:18:15:45 -0700]</td>
<td>&quot;set&quot;</td>
<td>foo</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>172.25.135.205:55438</td>
<td>[09/Jul/2012:18:15:46 -0700]</td>
<td>&quot;get&quot;</td>
<td>foo</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>172.25.135.205:55438</td>
<td>[09/Jul/2012:18:16:05 -0700]</td>
<td>&quot;set&quot;</td>
<td>bar</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>172.25.135.205:55438</td>
<td>[09/Jul/2012:18:16:09 -0700]</td>
<td>&quot;incr&quot;</td>
<td>bar</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

....
single producer, consumer

worker1

write ptr

read ptr

worker 2

write ptr

read ptr

logger

...
Twemproxy (nutcracker)

Running in production since Nov ’11

Supports memcached and redis

code: github.com/twitter/twemproxy
Motivation

\[ m >> n \Rightarrow 20mn \]
Deployed as Local Proxy

m >> n => m
Twemproxy

get k1

get k2

delete k3

delete k3, get k2, get k1

Friday, November 30, 12
Pipelining

time

get k1
get k2
delete k3

get k1
delete k3

get k2
Twemproxy in Production
Many thousands machines
10 - 20 server pools per instance
Each instance typically handles:
- few hundred client connections
- proxies to few thousands servers
Eg: 60K -> 3K connections
~2K rps, 200 KB/sec (req), 1MB/sec (rsp)
Why Proxy?

Persistent server connections
faster client restarts
filter close from client

Protocol pipelining
Enables simple and dumb clients
Hides semantics of underlying cache pool
Dynamic configuration
Why not Proxy?

Extra network hop

Tradeoff latency for throughput

Pipelining is your friend
What did we learn?

Hide caches behind abstraction layer

Indirection (proxies) enables horizontal scaling

Proxies add overhead and extra network hop

Minimize network hops by colocating proxies next to server / clients

Use pipelining to overcome additional overhead
New System Characteristics

Predictable worst case latency

Replicated

Read my Write

Eventually consistent

Use case: read volume >> write volume
key/value scheme

key = (outer-key, inner-key)

struct value {
    map<short, binary> fields = {};
    map<short, long> fieldTimestamps = {};
}

Friday, November 30, 12
Indirection and Colocation
Horizontal Scaling
Putting it all together

2 replicas

(1) (2) (3) (4)
Putting it all together

2 in-memory replicas

Persistent Store

Pub/Sub

(1) (2) (3) (4) (5) (5')
Questions?