Strategy: Break Up the Memcache Dog Pile

Friday, August 7, 2009 at 1:44AM

Todd Hoff in Django, Memcached, RoR, Strategy

Update: Asynchronous HTTP cache validations. A proposed HTTP caching extension: *if your application can afford to show slightly out of date content, then stale-while-revalidate can guarantee that the user will always be served directly from the cache, hence guaranteeing a consistent response-time user-experience.*

Caching is like aspirin for headaches. Head hurts: pop a 'sprin. Slow site: add caching. Facebook must have a lot of headaches because they popped 805 memcached servers between 10,000 web servers and 1,800 MySQL servers and they reportedly have a 99% cache hit rate. But what's the best way for you to cache for your application? It's a remarkably complex and rich topic. Alexey Kovyrin talks about one common caching problem called the Dog Pile *Effect* in Dog-pile Effect and How to Avoid it with Ruby on Rails. Glenn Franxman also has a Django solution in MintCache.

Data is usually cached because it's too expensive to calculate for every hit. Maybe it's a gnarly SQL query you want to avoid and a little stale data is OK. Or maybe the amount of data you have is simply larger than physical memory on any one machine. Or maybe you have the temerity to write to your database and cause its cache to flush so database caching isn't sufficient at a certain level of scale.

Typical examples are for caching article vote counts, comment threads, and event streams. One familiar example that bit me hard is displaying the the top N blog articles. Do you want to scan through your entire access log table for every page display? Absolutely not. Especially when the nightly backups are going on and the network is very slow. Not good :-) Yet you still want to update the results every X minutes so the stats stay fresh.

Data freshness requires a refrigeration truck or an expiry time on your cache entry that causes stats to be periodically recalculated. Now, what happens when your cached data expires and a 1000 requests simultaneously try to recalculate the expensive to calculate data? Database load spikes and the world nearly ends. And since memcached operations are not atomic it's possible stale data could be cached and

you'll serve stale data. Which kind of defeats of the purpose of taking load off the data while providing accurate data. So, how do you unpile the dogs?

No Expire Solution

If cache items never expire then there can never be a recalculation storm. Then how do you update the data? Use cron to periodically run the calculation and populate the cache. Take the responsibility for cache maintenance out of the application space. This approach can also be used to pre-warm the the cache so a newly brought up system doesn't peg the database.

The problem is the solution doesn't always work. Memcached can still evict your cache item when it starts running out of memory. It uses a LRU (least recently used) policy so your cache item may not be around when a program needs it which means it will have to go without, use a local cache, or recalculate. And if we recalculate we still have the same piling on issues.

This approach also doesn't work well for item specific caching. It works for globally calculated items like top N posts, but it doesn't really make sense to periodically cache items for user data when the user isn't even active. I suppose you could keep an active list to get around this limitation though.

Stale Date Solution

This solution introduces a stale date in addition to the expiration date. Glen describes it as:

> The first client to request data past the stale date is asked to refresh the data, while subsequent requests are given the stale but not-yet-expired data as if it were fresh, with the understanding that it will get refreshed in a 'reasonable' amount of time by that initial request

In the memcached FAQ a one key approach is described:

- Set the cache item expire time way out in the future.
- Embed the "real" timeout serialized with the value. For example, set the item to timeout in 24 hours, but the embedded timeout might be five minutes in the future.

- On a get from the cache determine if the stale timeout expired and on expiry immediately set a time in the future and re-store the data as is. This closes down the window of risk.
- Fetch data from the DB and update the cache with the latest value.

Alexey describes a different two key approach:

- Create two keys in memcached: MAIN key with expiration time a bit higher than normal + a STALE key which expires earlier.
- On a get read STALE key too. If the stale has expired, re-calculate and set the stale key again.

I dislike embedding meta data with data so I like Alexey's approach a bit better, even though it doubles the key space.

None of these options prevent the problem for ever happening, but they do greatly reduce the failure window for relatively little cost.

Related Articles

- Memcached Tag at High Scalability
- Caching Makes Your Brain Explode by Craig Ambrose.
- The Secret to Memcached by Tobias Lütke.
- Memcached FAQ.
- Dog-pile Effect and How to Avoid it with Ruby on Rails memcache-client Patch by Alexey Kovyrin.
- MintCache by Glenn Franxman.
- Advanced Rails Caching.. on the Edge by Aaron Batalion.
