Why cache invalidation is actually hard

Ka Wai Cheung, November 2023

We all know the famous phrase by Phil Karlton, “There are only two hard things in Computer Science: cache invalidation and naming things.” If you’ve done any kind of difficult programming, you know why naming is hard. But what about cache invalidation? No one seems to explain why that’s hard. If they do, it’s often about the actual hardware part of the problem…but that’s not why I find it hard. So, what follows is my reason.

The reason I find cache invalidation difficult is because oftentimes the data I am caching can be updated within the application from many different sources, and it's easy to forget what pieces of cached data need to be invalidated when updates from a seemingly disparate place happen.

For example, I cache a `UserProfile` object that I originally pulled from a single SQL query and hydrate into the object. The `UserProfile` contains name, email, number of projects assigned (via joining on a project memberships table and getting a `COUNT`). The key for that object might be something like `UserProfile-{UserID}`. Now, every time there's a case where any of those values might change, I need to be able to invalidate that cache key.

For name and email, this is pretty straightforward. The user updates their name or email, I invalidate that record. But what about the "number of projects assigned" bit? Here are some of the events that require me to invalidate that record:

1. When an admin adds the user to a project.
2. When an admin removes that user from a project.
3. When that project is archived or deleted.
4. When that project is reactivated (if it was archived).

#1 and #2 are again fairly straightforward as at the place that happens, you have the users ID at hand, but #3 and #4 need a bit more work. When the project's accessibility changes, I also have to grab *all* the current members of those projects and then invalidate each user's cached profile since all of their "number of projects assigned" values have been downgraded or upgraded.

You can extrapolate this problem to far more complex objects that have even more data entangled in other areas of the app and see that the amount of work you have to do to ensure valid cached records gets rapidly more complex.

The antidote may be to cache more simple record types. But then again, usually the more complex a record is to manifest from its data sources, the more beneficial a cached record is to the system.

The art of it all is deciding what makes sense enough to cache regularly—data that doesn't change often, that has a meaningful performance hit to the system when accessed directly, and that isn't a pain to invalidate (especially as the system grows in functionality). It's hard to foresee the last bit, so as systems grow, the chance for invalid cached data grows too.