Caching Best Practices

How to apply caching

Caching is applicable to a wide variety of use cases, but fully exploiting caching requires some planning. When deciding whether to cache a piece of data, consider the following questions:

- Is it safe to use a cached value? The same piece of data can have different consistency requirements in different contexts. For example, during online checkout, you need the authoritative price of an item, so caching might not be appropriate. On other pages, however, the price might be a few minutes out of date without a negative impact on users.
- Is caching effective for that data? Some applications generate access patterns that are not suitable for caching—for example, sweeping through the key space of a large dataset that is changing frequently. In this case, keeping the cache up to date could offset any advantage caching could offer.
- Is the data structured well for caching? Simply caching a database record can often be enough to

offer significant performance advantages. However, other times, data is best cached in a format that combines multiple records together. Because caches are simple key-value stores, you might also need to cache a data record in multiple different formats, so you can access it by different attributes in the record.

You don’t need to make all of these decisions up front. As you expand your usage of caching, keep these guidelines in mind when deciding whether to cache a given piece of data.

Caching design patterns

Lazy caching

Lazy caching, also called lazy population or cache-aside, is the most prevalent form of caching. Laziness should serve as the foundation of any good caching strategy. The basic idea is to populate the cache only when an object is actually requested by the application. The overall application flow goes like this:

1. Your app receives a query for data, for example the top 10 most recent news stories.
2. Your app checks the cache to see if the object is in cache.
3. If so (a cache hit), the cached object is returned, and the call flow ends.
4. If not (a cache miss), then the database is queried for the object. The cache is populated, and the object is returned.

This approach has several advantages over other methods:

- The cache only contains objects that the application actually requests, which helps keep the cache size manageable. New objects are only added to the cache as needed. You can then manage your cache memory passively, by simply letting the engine you are using evict the least-accessed keys as your cache fills up, which it does by default.
- As new cache nodes come online, for example as your application scales up, the lazy population method will automatically add objects to the new cache nodes when the application first requests them.
- Cache expiration is easily handled by simply deleting the cached object. A new object will be fetched from the database the next time it is requested.
- Lazy caching is widely understood, and many web and app frameworks include support out of the box.

Here is an example of lazy caching in Python pseudocode:

```
Python

# Python

def get_user(user_id):

    # Check the cache

    record = cache.get(user_id)

    if record is None:

       # Run a DB query

       record = db.query("select * from users where id = ?",user_id)

       # Populate the cache

       cache.set(user_id, record)

    return record

# App code

user = get_user(17)
```

You can find libraries in many popular programming frameworks that encapsulate this pattern. But regardless of programming language, the overall approach is the same.

You should apply a lazy caching strategy anywhere in your app where you have data that is going to be read often, but written infrequently. In a typical web or mobile app, for example, a user's profile rarely changes, but is accessed throughout the app. A person might only update his or her profile a few times a year, but the profile might be accessed dozens or hundreds of times a day, depending on the user. Popular technologies that are used for caching like Memcached and Redis will automatically evict the less frequently used cache keys to free up memory if you set an eviction policy. Thus you can apply lazy caching liberally with little downside.

Write-through

In a write-through cache, the cache is updated in real time when the database is updated. So, if a user updates his or her profile, the updated profile is also pushed into the cache. You can think of this as being proactive to avoid unnecessary cache misses, in the case that you have data that you absolutely know is going to be accessed. A good example is any type of aggregate, such as a top 100 game leaderboard, or the top 10 most popular news stories, or even recommendations. Because this data is typically updated by a specific piece of application or background job code, it's straightforward to update the cache as well.

The write-through pattern is also easy to demonstrate in pseudocode:

```
Python

 # Python

 def save_user(user_id, values):

     # Save to DB

     record = db.query("update users ... where id = ?", user_id, values)

     # Push into cache

     cache.set(user_id, record)

     return record

 # App code

 user = save_user(17, {"name": "Nate Dogg"})
```

This approach has certain advantages over lazy population:

- It avoids cache misses, which can help the application perform better and feel snappier.
- It shifts any application delay to the user updating data, which maps better to user expectations. By contrast, a series of cache misses can give a user the impression that your app is just slow.
- It simplifies cache expiration. The cache is always up-to-date.

However, write-through caching also has some disadvantages:

- The cache can be filled with unnecessary objects that aren't actually being accessed. Not only could this consume extra memory, but unused items can evict more useful items out of the cache.
- It can result in lots of cache churn if certain records are updated repeatedly.
- When (not if) cache nodes fail, those objects will no longer be in the cache. You need some way to repopulate the cache of missing objects, for example by lazy population.

As might be obvious, you can combine lazy caching with write-through caching to help address these issues, because they are associated with opposite sides of the data flow. Lazy caching catches cache misses on reads, and write-through caching populates data on writes, so the two approaches complement each other. For this reason, it's often best to think of lazy caching as a foundation that you can use throughout your app, and write-through caching as a targeted optimization that you apply to specific situations.

Time-to-live

Cache expiration can get really complex really quickly. In our previous examples, we were only operating on a single user record. In a real app, a given page or screen often caches a whole bunch of different stuff at once—profile data, top news stories, recommendations, comments, and so forth, all of which are being updated by different methods.

Unfortunately, there is no silver bullet for this problem, and cache expiration is a whole arm of computer science. But there are a few simple strategies that you can use:

- Always apply a time to live (TTL) to all of your cache keys, except those you are updating by write-through caching. You can use a long time, say hours or even days. This approach catches application bugs, where you forget to update or delete a given cache key when updating the underlying record. Eventually, the cache key will auto-expire and get refreshed.
- For rapidly changing data such as comments, leaderboards, or activity streams, rather than adding write-through caching or complex expiration logic, just set a short TTL of a few seconds. If you have a database query that is getting hammered in production, it's just a few lines of code to add a cache key with a 5 second TTL around the query. This code can be a wonderful Band-Aid to keep your application up and running while you evaluate more elegant solutions.
- A newer pattern, Russian doll caching, has come out of work done by the Ruby on Rails team. In this pattern, nested records are managed with their own cache keys, and then the top-level resource is a collection of those cache keys. Say you have a news webpage that contains users, stories, and comments. In this approach, each of those is its own cache key, and the page queries each of those keys respectively.
- When in doubt, just delete a cache key if you're not sure whether it's affected by a given database update or not. Your lazy caching foundation will refresh the key when needed. In the meantime, your database will be no worse off than it was without caching.

For a good overview of cache expiration and Russian doll caching, refer to The performance impact of "Russian doll" caching, a post in the Basecamp Signal vs Noise blog.

Evictions

Evictions occur when memory is over filled or greater than maxmemory setting in the cache, resulting into the engine to select keys to evict in order to manage its memory. The keys that are chosen are based on the eviction policy that is selected.

By default, Amazon ElastiCache for Redis sets the volatile-lru eviction policy to your Redis cluster. This policy selects the least recently used keys that have an expiration (TTL) value set. Other eviction policies are available can be applied as configurable maxmemory-policy parameter. Eviction policies can be summarized as the following:

allkeys-lfu: The cache evicts the least frequently used (LFU) keys regardless of TTL set
allkeys-lru: The cache evicts the least recently used (LRU) regardless of TTL set
volatile-lfu: The cache evicts the least frequently used (LFU) keys from those that have a TTL set
volatile-lru: The cache evicts the least recently used (LRU) from those that have a TTL set
volatile-ttl: The cache evicts the keys with shortest TTL set
volatile-random: The cache randomly evicts keys with a TTL set
allkeys-random: The cache randomly evicts keys regardless of TTL set
no-eviction: The cache doesn’t evict keys at all. This blocks future writes until memory frees up.

A good strategy in selecting an appropriate eviction policy is to consider the data stored in your cluster and the outcome of keys being evicted.
Generally, LRU based policies are more common for basic caching use-cases, but depending on your objectives, you may want to leverage a TTL or Random based eviction policy if that better suits your requirements.

Also, if you are experiencing evictions with your cluster, it is usually a sign that you need to scale up (use a node that has a larger memory footprint) or scale out (add additional nodes to the cluster) in order to accommodate the additional data. An exception to this rule is if you are purposefully relying on the cache engine to manage your keys by means of eviction, also referred to an LRU cache.

The thundering herd

Also known as dog piling, the thundering herd effect is what happens when many different application processes simultaneously request a cache key, get a cache miss, and then each hits the same database query in parallel. The more expensive this query is, the bigger impact it has on the database. If the query involved is a top 10 query that requires ranking a large dataset, the impact can be a significant hit.

One problem with adding TTLs to all of your cache keys is that it can exacerbate this problem. For example, let's say millions of people are following a popular user on your site. That user hasn't updated his profile or published any new messages, yet his profile cache still expires due to a TTL. Your database might suddenly be swamped with a series of identical queries.

TTLs aside, this effect is also common when adding a new cache node, because the new cache node's memory is empty. In both cases, the solution is to prewarm the cache by following these steps:

1. Write a script that performs the same requests that your application will. If it's a web app, this script can be a shell script that hits a set of URLs.

2. If your app is set up for lazy caching, cache misses will result in cache keys being populated, and the new cache node will fill up.
3. When you add new cache nodes, run your script before you attach the new node to your application. Because your application needs to be reconfigured to add a new node to the consistent hashing ring, insert this script as a step before triggering the app reconfiguration.
4. If you anticipate adding and removing cache nodes on a regular basis, prewarming can be automated by triggering the script to run whenever your app receives a cluster reconfiguration event through Amazon Simple Notification Service (Amazon SNS).

Finally, there is one last subtle side effect of using TTLs everywhere. If you use the same TTL length (say 60 minutes) consistently, then many of your cache keys might expire within the same time window, even after prewarming your cache. One strategy that's easy to implement is to add some randomness to your TTL:

```
Python

ttl = 3600 + (rand() * 120)  /* +/- 2 minutes */
```

The good news is that only sites at large scale typically have to worry about this level of scaling problem. It's good to be aware of, but it's also a good problem to have.

Cache (almost) everything

Finally, it might seem as if you should only cache your heavily hit database queries and expensive calculations, but that other parts of your app might not benefit from caching. In practice, in-memory caching is widely useful, because it is much faster to retrieve a flat cache key from memory than to perform even the most highly optimized database query or remote API call. Just keep in mind that cached data is stale data by definition, meaning there may be cases where it’s not appropriate, such as accessing an item’s price during online checkout. You can monitor statistics like cache misses to see whether your cache is effective.

Caching technologies

The most popular caching technologies are in the in-memory Key-Value category of NoSQL databases. An in-memory key-value store is a NoSQL database optimized for read-heavy application workloads (such as social networking, gaming, media sharing and Q&A portals) or compute-intensive workloads (such as a recommendation engine). Two key benefits make in-memory key-value stores popular as caching solutions – speed and simplicity. Key value stores don’t have complex query or aggregation logic, making queries fast. The in-memory key-value stores are especially fast due to them utilizing memory rather than slower disk. In addition, their simplicity makes them easy to master and utilize. There are numerous key-value technologies available on the market, many of them could be used as caching solutions. Two highly popular in-memory key-value stores are Memcached and Redis. AWS allows running both these engines in a fully managed fashion through Amazon ElastiCache.

Get started with Amazon ElastiCache

It's easy to get started with caching in the cloud with a fully-managed service like Amazon ElastiCache. It removes the complexity of setting up, managing and administering your cache, and frees you up to focus on what brings value to your organization. Sign up today for Amazon ElastiCache.

More Caching Resources

Technical Whitepaper on Database Caching Strategies Using Redis

Technical Whitepaper on In-memory Caching

Amazon CloudFront

Ready to build?

Get started with Amazon ElastiCache

Have more questions?

Contact us