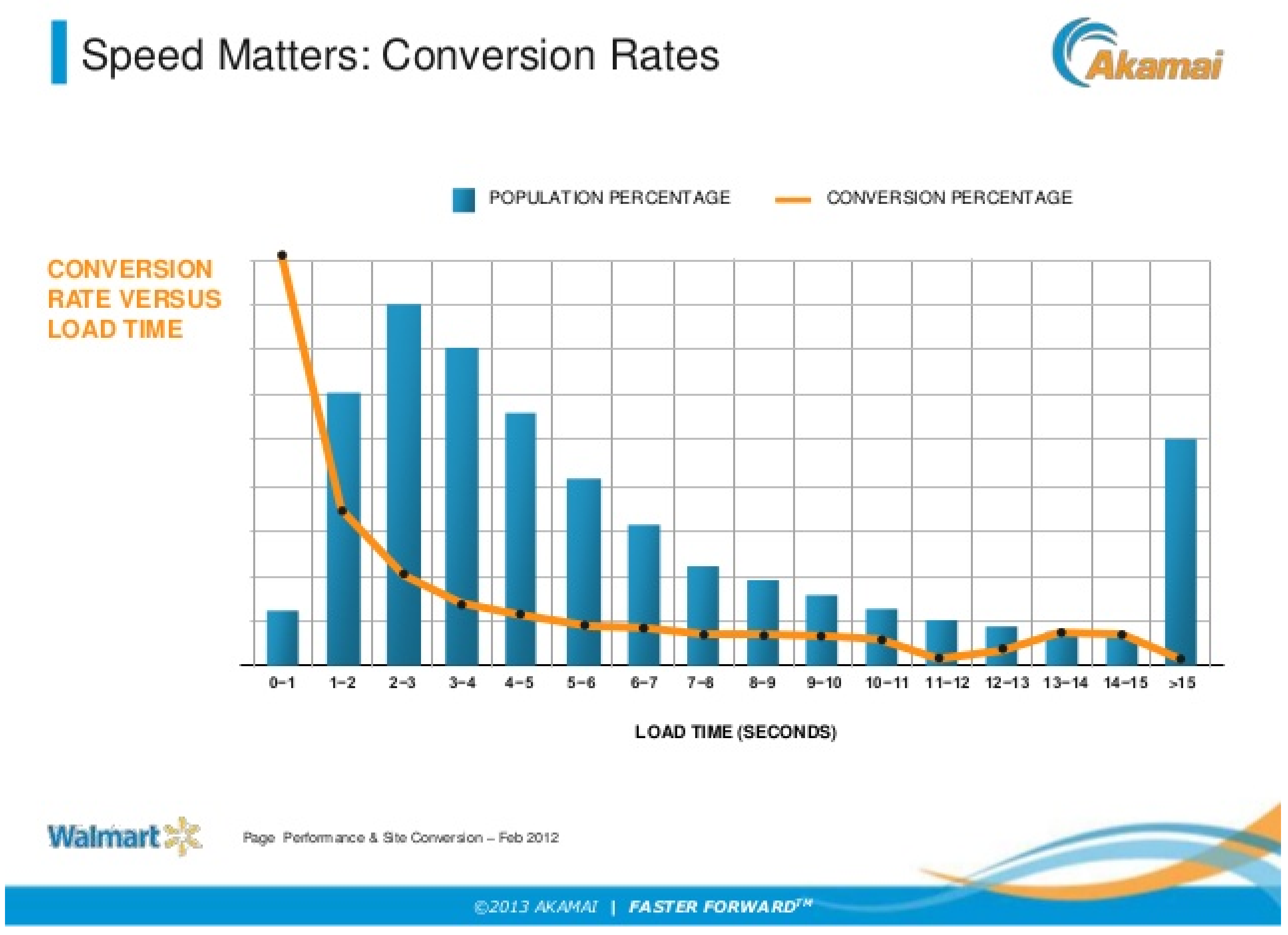
Caching Strategies and How to Choose the Right One

Umer Mansoor Follow
Aug 11, 2017 · 7 mins read

1.8k
Shares

Caching is one of the easiest ways to increase system performance. Databases can be slow (yes even the NoSQL ones) and as you already know, speed is the name of the game.



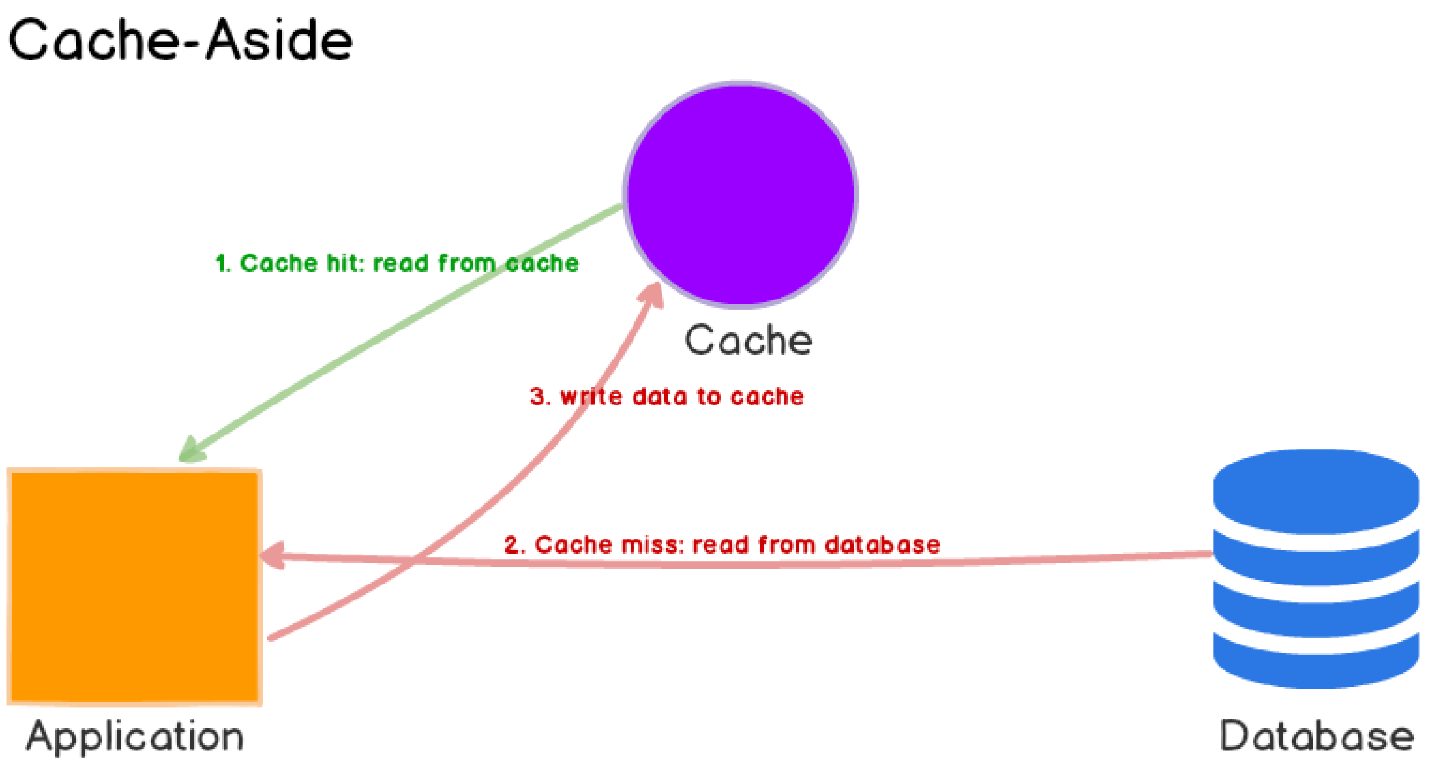
If done *right*, caches can reduce response times, decrease load on database, and save costs. There are several strategies and choosing the *right* one can make a big difference. Your caching strategy depends on the data and **data access patterns**. In other words, how the data is written and read. For example:

- is the system write heavy and reads less frequently? (e.g. time based logs)
- is data written once and read multiple times? (e.g. User Profile)
- is data returned always unique? (e.g. search queries)

A caching strategy for Top-10 leaderboard system for mobile games will be very different than a service which aggregates and returns user profiles. Choosing the right caching strategy is the key to improving performance. Let's take a quick look at various caching strategies.

Cache-Aside

This is perhaps the most commonly used caching approach, at least in the projects that I worked on. The cache sits on the *side* and the application directly talks to both the cache and the database.


Cache-Aside
1. Cache hit: read from cache
Cache
3. write data to cache
2. Cache miss: read from database
Application
Database

Here's what's happening:

1. The application first checks the cache.
2. If the data is found in cache, we've *cache hit*. The data is read and returned to the client.
3. If the data is **not found** in cache, we've *cache miss*. The application has to do some **extra work**. It queries the database to read the data, returns it to the client and stores the data in cache so the subsequent reads for the same data results in a cache hit.

Use Cases, Pros and Cons

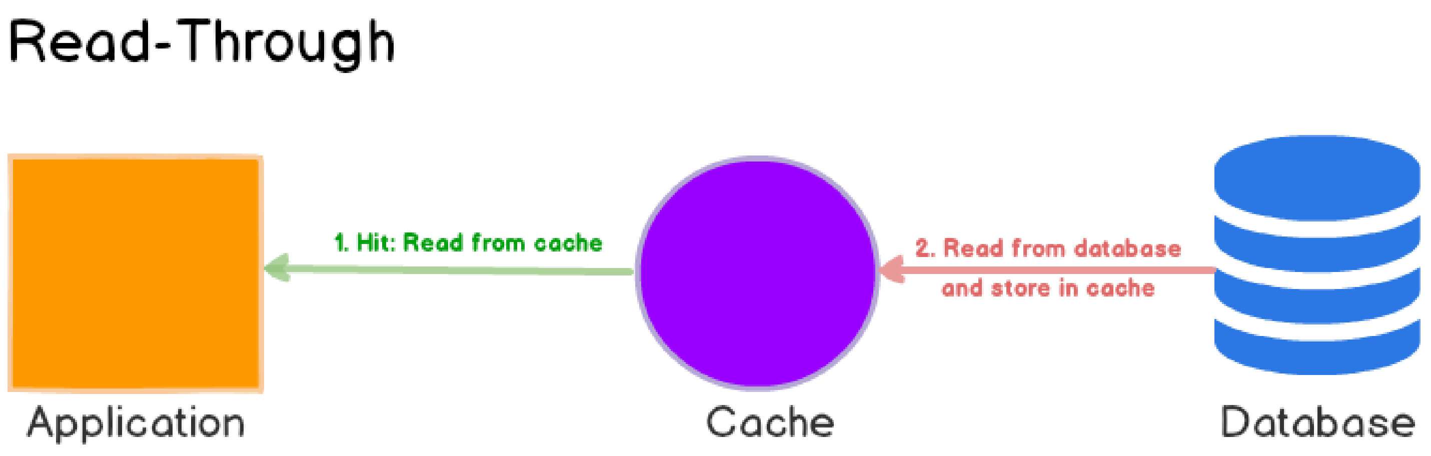
Cache-aside caches are usually general purpose and work best for **read-heavy workloads**. *Memcached* and *Redis* are widely used. Systems using cache-aside are **resilient to cache failures**. If the cache cluster goes down, the system can still operate by going directly to the database. (Although, it doesn't help much if cache goes down during peak load. Response times can become terrible and in worst case, the database can stop working.)

Another benefit is that the data model in cache can be different than the data model in database. E.g. the response generated as a result of multiple queries can be stored against some request id.

When cache-aside is used, the most common write strategy is to write data to the database directly. When this happens, cache may become inconsistent with the database. To deal with this, developers generally use time to live (TTL) and continue serving stale data until TTL expires. If data freshness must be guaranteed, developers either **invalidate the cache entry** or use an appropriate write strategy, as we'll explore later.

Read-Through Cache

Read-through cache sits in-line with the database. When there is a cache miss, it loads missing data from database, populates the cache and returns it to the application.



Both cache-aside and read-through strategies load data **lazily**, that is, only when it is first read.

Use Cases, Pros and Cons

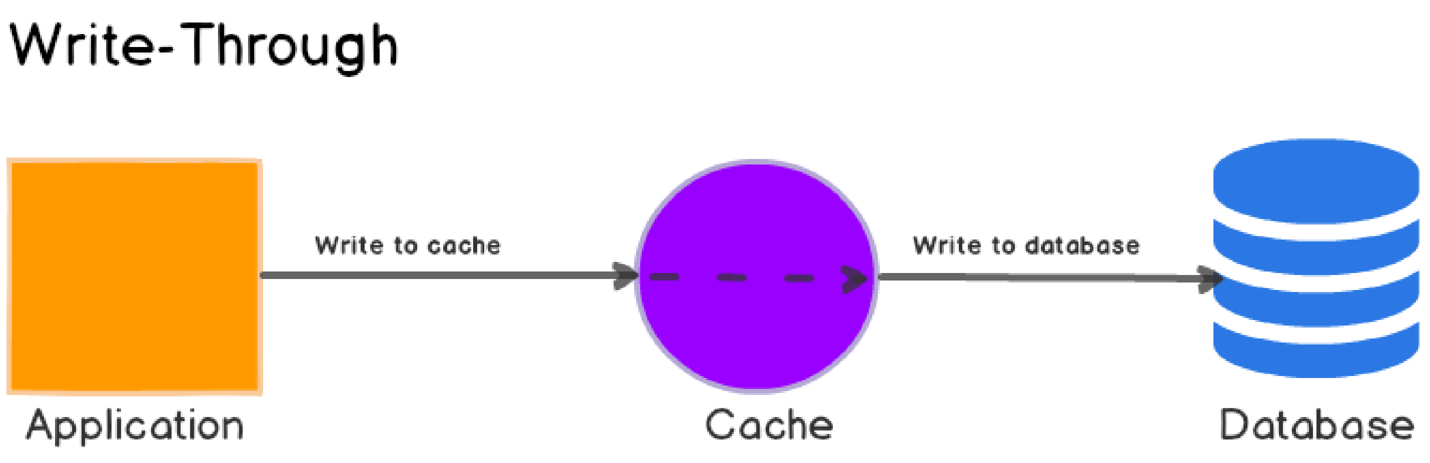
While read-through and cache-aside are very similar, there are at least two key differences:

1. In cache-aside, the application is responsible for fetching data from the database and populating the cache. In read-through, this logic is usually supported by the library or stand-alone cache provider.
2. Unlike cache-aside, the data model in read-through cache cannot be different than that of the database.

Read-through caches work best for **read-heavy** workloads when the same data is requested many times. For example, a news story. The disadvantage is that when the data is requested the first time, it always results in cache miss and incurs the extra penalty of loading data to the cache. Developers deal with this by '*warming*' or 'pre-heating' the cache by issuing queries manually. Just like cache-aside, it is also possible for data to become inconsistent between cache and the database, and solution lies in the write strategy, as we'll see next.

Write-Through Cache

In this write strategy, data is first written to the cache and then to the database. The cache sits in-line with the database and writes always go *through* the cache to the main database.



Use Cases, Pros and Cons

On its own, write-through caches don't seem to do much, in fact, they introduce extra write latency because data is written to the cache first and then to the main database. But when paired with read-through caches, we get all the benefits of read-through and we also get data consistency guarantee, freeing us from using cache invalidation techniques.

DynamoDB Accelerator (DAX) is a good example of read-through / write-through cache. It sits inline with DynamoDB and your application. Reads and writes to DynamoDB can be done through DAX. (Side note: If you are planning to use DAX, please make sure you familiarize yourself with its data consistency model and how it interplays with DynamoDB.)

Write-Around

Here, data is written directly to the database and only the data that is read makes it way into the cache.

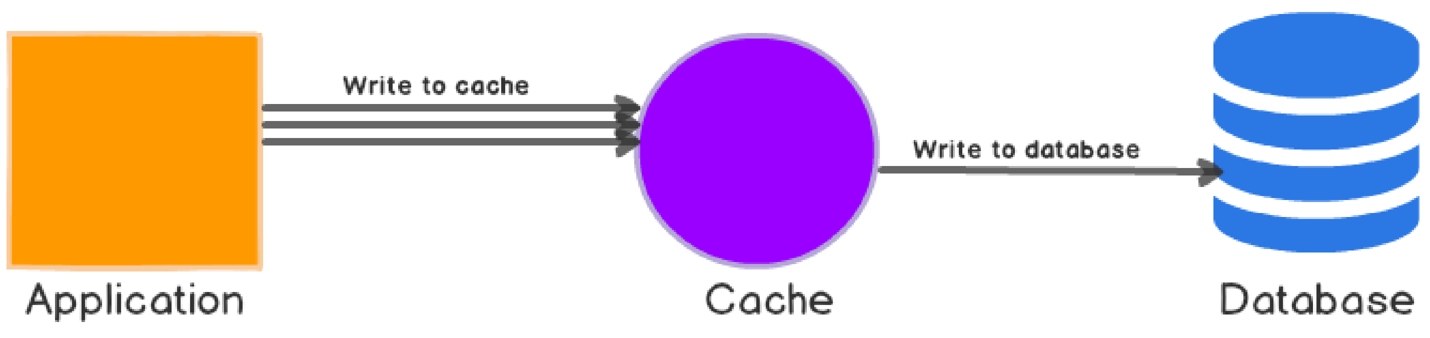
Use Cases, Pros and Cons

Write-around can be combine with read-through and provides good performance in situations where data is written once and read less frequently or never. For example, real-time logs or chatroom messages. Likewise, this pattern can be combined with cache-aside as well.

Write-Back

Here, the application writes data to the cache which acknowledges immediately and after some *delay*, it writes the data *back* to the database.

Write-Back



This is sometimes called write-behind as well.

Use Cases, Pros and Cons

Write back caches improve the write performance and are good for **write-heavy** workloads. When combined with read-through, it works good for mixed workloads, where the most recently updated and accessed data is always available in cache.

It's resilient to database failures and can tolerate some database downtime. If batching or coalescing is supported, it can reduce overall writes to the database, which decreases the load and **reduces costs**, if the database provider charges by number of requests e.g. DynamoDB. Keep in mind that **DAX is write-through** so you won't see any reductions in costs if your application is write heavy. (When I first heard of DAX, this was my first question - DynamoDB can be very expensive, but damn you Amazon.)

Some developers use Redis for both cache-aside and write-back to better absorb spikes during peak load. The main disadvantage is that if there's a cache failure, the data may be permanently lost.

Most relational databases storage engines (i.e. InnoDB) have write-back cache enabled by default in their internals. Queries are first written to memory and eventually flushed to the disk.

Summary

In this post, we explored different caching strategies and their pros and cons. In practice, carefully evaluate your goals, understand data access (read/write) patterns and choose the best strategy or a combination.

What happens if you choose wrong? One that doesn't match your goals or access patterns? You may introduce additional latency, or at the very least, not see the *full benefits*. For example, if you choose *write-through/read-through* when you actually should be using *write-around/read-through* (written data is accessed less frequently), you'll have useless junk in your cache. Arguably, if the cache is big enough, it may be fine. But in many real-world, high-throughput systems, when memory is never big enough and server costs are a concern, the right strategy, matters.