

Caching Strategies

Master System Design

Progress 22/130 chapters

Search topics...

Networking 8/9

API Fundamentals 5/10

Databases & Storage 0/12

Database Scaling Techniques 0/8

Caching 1/6

What is Caching?

Read-Through vs Write-Through Cache

Caching Strategies

Ashish Pratap Singh



5 min read

Caching is a powerful technique to **reduce latency** and **improve system performance**.

There are several **caching strategies**, depending on what a system needs - whether the focus is on optimizing for **read-heavy** workloads, **write-heavy** operations, or ensuring **data consistency**.

In this article, we'll cover the **5 most common caching strategies** that frequently come up in **system design discussions** and widely used in **real-world applications**.

1. Read Through

In the **Read Through** strategy, the cache acts as an intermediary between the application and the database.

★ Get Premium
Subscribe to unlock full access to all premium content

Subscribe Now

Reading Progress 0%

On this page

1. Read Through

2. Cache Aside

3. Write Through

4. Write Around

5. Write Back

Summary

- Cache Eviction Policies
- Distributed Caching
- Content Delivery Network (CDN)



Asynchronous Communications

0/4 ▾



Tradeoffs

0/9 ▾



Distributed System Concepts

0/10 ▾



Architectural Patterns

0/5 ▾



Microservices

0/6 ▾



Big Data Processing

0/5 ▾



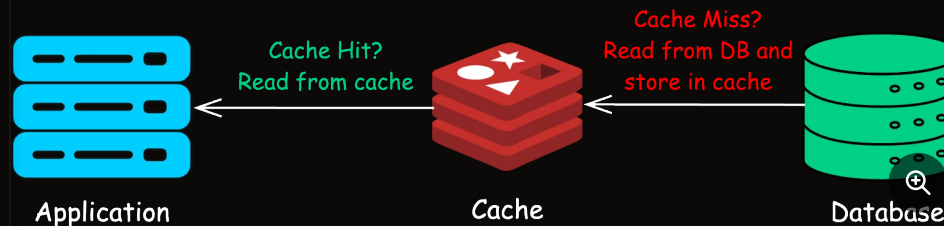
Observability

0/3 ▾

When the application requests data, it first looks in the cache.

If data is available (**cache hit**), it's returned to the application.

If the data is not available (**cache miss**), the cache itself is responsible for fetching the data from the database, storing it, and returning it to the application.



This approach **simplifies application logic** because the application does not need to handle the logic for fetching and updating the cache.

The cache itself handles both reading from the database and storing the requested data automatically. This minimizes unnecessary data in the cache and ensures that frequently accessed data is readily available.

For **cache hits**, Read Through provides **low-latency** data access.

But for **cache misses**, there is a potential **delay** while the cache queries the database and stores the data. This can

result in higher latency during initial reads.

To prevent the cache from serving stale data, a **time-to-live (TTL)** can be added to cached entries. TTL automatically expires the data after a specified duration, allowing it to be reloaded from the database when needed.

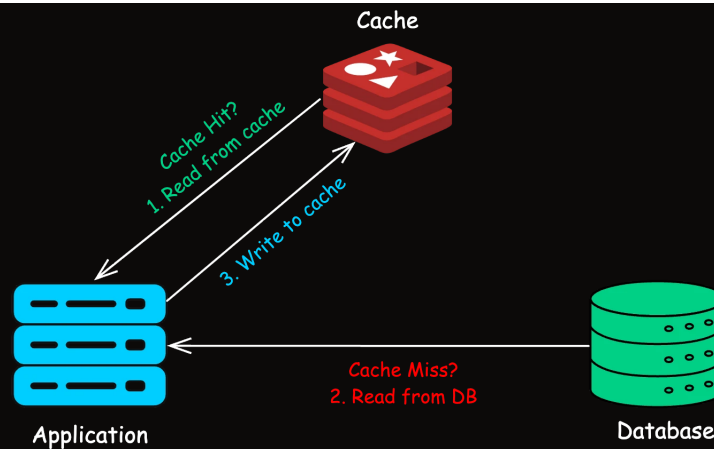
Read Through caching is best suited for **read-heavy applications** where data is accessed frequently but updated less often, such as content delivery systems (CDNs), social media feeds, or user profiles.

2. Cache Aside

Cache Aside, also known as "**Lazy Loading**", is a strategy where the **application code** handles the interaction between the cache and the database. The data is loaded into the cache only when needed.

The application first checks the cache for data. If the data exists in cache (**cache hit**), it's returned to the application.

If the data isn't found in cache (**cache miss**), the application retrieves it from the database (or the primary data store), then loads it into the cache for subsequent requests.



The cache acts as a "**sidecar**" to the database, and it's the responsibility of the application to manage when and how data is written to the cache.

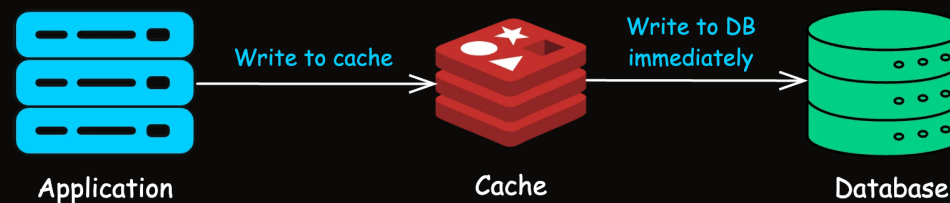
To avoid stale data, we can set a **time-to-live (TTL)** for cached data. Once the TTL expires, the data is automatically removed from the cache.

Cache Aside is perfect for systems where the **read-to-write** ratio is **high**, and data updates are infrequent. For example, in an e-commerce website, product data (like prices, descriptions, or stock status) is often read much more frequently than it's updated.

3. Write Through

In the **Write Through** strategy, every write operation is executed on both the cache and the database at the same time.

This is a **synchronous process**, meaning both the cache and the database are updated as part of the same operation, ensuring that there is no delay in data propagation.



This approach ensures that the cache and the database remain **synchronized** and the read requests from the cache will always return the **latest data**, avoiding the risk of serving stale data.

In a Write Through caching strategy, cache expiration policies (such as TTL) are generally not necessary. However, if you are concerned about cache memory usage, you can implement a TTL policy to remove infrequently accessed data after a certain time period.

The biggest advantage of Write Through is that it ensures strong **data consistency** between the cache and the database.

Since the cache always contains the latest data, **read oper-**

ations benefit from **low latency** because data can be directly retrieved from the cache.

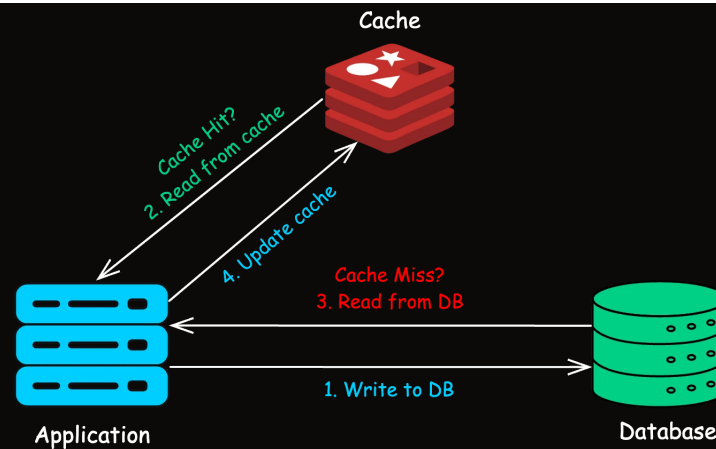
However, **write latency** can be **higher** due to the overhead of writing to both the cache and the database.

Write Through is ideal for **consistency-critical systems**, such as financial applications or online transaction processing systems, where the cache and database must always have the latest data.

4. Write Around

Write Around is a caching strategy where data is written directly to the database, bypassing the cache.

The cache is only updated when the data is requested later during a read operation, at which point the **Cache Aside** strategy is used to load the data into the cache.



This approach ensures that only **frequently accessed data** resides in the cache, preventing it from being polluted by data that may not be accessed again soon.

It keeps the cache clean by avoiding unnecessary data that might not be requested after being written.

Writes are relatively **faster** because they only target the database and don't incur the overhead of writing to the cache.

TTL can be used to ensure that data does not remain in the cache indefinitely. Once the TTL expires, the data is removed from the cache, forcing the system to retrieve it from the database again if needed.

Write Around caching is best used in **write-heavy systems** where data is frequently written or updated,

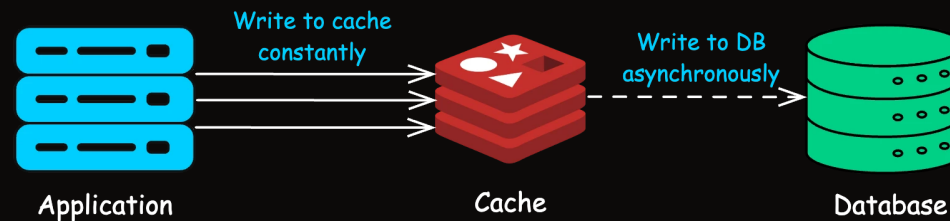
but **not immediately or frequently read** such as logging systems.

5. Write Back

In the **Write Back** strategy, data is first written to the cache and then **asynchronously** written to the database at a later time.

This strategy focuses on **minimizing write latency** by deferring database writes.

This deferred writing means that the cache acts as the primary storage during write operations, while the database is updated periodically in the background.



The key advantage of Write Back is that it significantly **reduces write latency**, as writes are completed quickly in the cache, and the database updates are delayed or batched.

However, with this approach, there is a risk of **data loss** if

the cache fails before the data has been written to the database.

This can be mitigated by using persistent caching solutions like **Redis with AOF (Append Only File)**, which logs every write operation to disk, ensuring data durability even if the cache crashes.

Write Back doesn't require invalidation of cache entries, as the cache itself is the source of truth during the write process.


Write Back caching is ideal for **write-heavy** scenarios where write operations need to be **fast** and **frequent**, but **immediate consistency** with the database is not critical, such as logging systems and social media feeds.


Summary


Choosing the right caching strategy depends on your system's specific requirements.


Here's a tabular summary:

< Prev: Read-Through vs Write-Thr...

 Take Notes

 Star

 Mark as Complete

 Ask AI

New

Next: Cache Eviction Policies >