**Every Caching Strategy Explained in 5 Minutes**

**Never fail a system design interview again.**

Caching is one of the simplest concepts for devs to know.

**The Goal:** Make things faster and reduce load on primary data stores (like databases). Caches offer quicker access and shield your backend from repetitive requests.

**The Main Strategies**

**1. Cache-Aside (Lazy Loading)**

This is arguably the most common approach you’ll encounter. With Cache-Aside, your application code takes direct responsibility for managing the cache. When data is needed, the application first checks if it’s in the cache. If there’s a **cache hit**, the data is returned immediately. On a **cache miss**, the application fetches the data from the primary source (like your database), stores a copy in the cache for next time, and *then* returns it.

Imagine fetching a user’s profile page: the app checks the cache for user:123. If it’s not there, it queries the DB, places the result in the cache under user:123, and proceeds.

**Use it when:**

* You primarily deal with read-heavy workloads.
* Occasional stale data is acceptable (if the DB changes without cache invalidation).
* You favour simplicity in the cache interaction logic within your application.

Request Data

Yes

No

Fetch Data

Store Data

Return Data

Legend

Application

Cache

Database

**Cache-Aside (Lazy Loading) Pattern**

Application

Cache

Database

Data Found?

Application

Application

Application

**2. Read-Through**

In a Read-Through strategy, the application interacts *only* with the cache for reads, treating it as the main data source. The magic happens behind the scenes: if the requested data isn’t in the cache (a miss), the cache itself is responsible for fetching it from the underlying database, storing it, and then returning it to the application. This simplifies your application code considerably, as it doesn’t need database-fetching logic for reads.

Think of a product catalog service using a cache library configured with a CacheLoader. The application simply calls cache.get(“product:xyz”), and the cache system handles the database interaction on a miss.

**Use it when:**

* Your workloads are read-heavy.
* You want to abstract data fetching logic away from the main application flow.
* Your chosen cache provider (like some libraries or managed services) explicitly supports this automatic data loading feature.

ApplicationCacheData Found?DatabaseRequest DataReturn DataFetch DataStore DataRead-Through Caching StrategyLegendApplicationCacheDatabaseData FlowReturn Data

**3. Write-Through**

Consistency is king with the Write-Through strategy. When your application needs to write or update data, it does so in two places: the cache *and* the database. The operation is only considered complete once *both* stores have successfully acknowledged the write. This guarantees that the cache is always consistent with the database, reducing the chance of serving stale data.

A critical update, like changing a user’s email address, is a prime candidate. The application ensures the new email is saved in both the cache and the database before confirming success. The trade-off is potentially higher write latency, as you’re waiting for two operations.

**Use it when:** Data consistency is paramount, you cannot tolerate discrepancies between the cache and the database, and slightly slower write performance is an acceptable trade-off.

1: Write Data

2: Write Data

✓ Cache Write

✓ DB Write

3: Success Response

Legend

Application

Cache

Database

Both writes must complete

**Write-Through Caching Pattern**

Application

Cache

Database

Both writes must complete

**4. Write-Behind (Write-Back)**

Need blazing fast writes? Write-Behind might be your answer. Here, the application writes data *only* to the cache, which acknowledges the write almost instantly. The cache then takes on the responsibility of asynchronously writing that data back to the database later, often after a short delay or by batching multiple writes together. This significantly improves write performance from the application’s perspective.

This is great for high-frequency updates like view counters, social media ‘likes’, or real-time game scores where speed is critical. However, there’s a risk: if the cache fails before the data is persisted to the database, that data could be lost.

**Use it when:** Write performance is the top priority, your application generates bursts of writes, and you can tolerate a small risk of data loss in case of cache failure before the asynchronous write completes.

1: Write Data

2: Immediate Success

Store in Queue

3: Async Write (delayed/batched)

Legend

Async Operation

Application

Cache

Database

Queue

**Write-Behind (Write-Back) Caching Pattern**

Application

Cache

Write Queue

Database

**5. Write-Around**

Sometimes, involving the cache during writes is unnecessary or even detrimental. The Write-Around strategy handles this by having the application write data directly to the database, completely bypassing the cache. Data only enters the cache when it’s subsequently *read* (typically using the Cache-Aside pattern for the read operation).

Consider bulk data imports or intensive logging. Writing this data straight to the database prevents flooding the cache with information that might not be accessed frequently or immediately, keeping the cache focused on hotter, more relevant data.

**Use it when:** You have write-heavy workloads where the data isn’t likely to be read soon after being written, and you want to avoid polluting the cache with potentially “cold” data.

Read Path

Request Data

Check

Yes

No

Fetch Data

Store

Return Data

Application

Cache

Data Found?

Application

Database

Application

Application

Write Path

Write Data

Application

Database

**Write-Around Caching Pattern**

**Choosing Wisely**

No single strategy is best. Choose based on your application’s needs:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strategy** | **Read Speed** | **Write Speed** | **Consistency** | **Cache Complexity** |
| Cache-Aside | Fast | Normal | Medium | Manual |
| Read-Through | Fast | Normal | Medium | Abstracted |
| Write-Through | Fast | Slower | High | Higher |
| Write-Behind | Fast | Fast | Low-Med | Higher |
| Write-Around | Normal | Fast | Medium | Simple |

**🧠 Quick Knowledge Check**

You’re designing a **real-time leaderboard service** for a mobile game with millions of daily active users. Each time a player finishes a match, their score is updated. These updates happen **frequently** and **must be fast**, so players see instant feedback on their rank. Occasionally, a score update might be lost (e.g., if a player closes the app mid-update), but **the system should not slow down** due to these rare edge cases. Eventually, all scores should be persisted to the database for long-term analytics.

Which caching strategy is most appropriate for the **score update logic**?

A) **Cache-Aside** – The application reads and writes directly to the database and manually updates the cache as needed.  
B) **Write-Through** – The application writes to the cache, and the cache immediately writes to the database before returning success.  
C) **Write-Behind (Write-Back)** – The application writes to the cache, and the cache queues updates to the database asynchronously.  
D) **Write-Around** – The application writes directly to the database, skipping the cache entirely; the cache is only populated on reads.

Answer

**C) Write-Behind (Write-Back)**

**Why?** Write-Behind is ideal for this use case because:

* **Performance is critical.** Players expect immediate feedback, and Write-Behind gives near-instant “write success” by updating the cache only.
* **High write volume.** The system handles millions of updates per day; batching writes to the database asynchronously is far more efficient than doing them one by one in real time.
* **Eventual consistency is acceptable.** If a score is briefly out of sync or even lost in rare cases (e.g., if the cache crashes before writing to the DB), it’s not catastrophic — the game logic and leaderboards can tolerate small inconsistencies in favor of speed.
* **Reduced database load.** Write-Behind helps prevent your DB from becoming a bottleneck during peak traffic.

**Why not the others?**

**D) Write-Around** ignores the cache on writes, leading to cold reads and delayed leaderboard updates.

**A) Cache-Aside** requires the app to manage reads/writes manually and offers no performance gain for heavy writes.

**B) Write-Through** guarantees consistency but adds latency to every write — too slow for real-time updates.